基于ES6，详细讲解ECMAScript语言规范

- 详解ECMAScript语言规范，启发读者领悟ES编程思想
- 揭示ES各个知识点之间的关联，深入理解ECMAScript语言
- 200多个示例代码+教学视频，快速掌握ECMAScript编程方法





ECMAScript 从零开始学（视频教学版）

王金柱　编著



清华大学出版社





ECMAScript 从零开始学（视频教学版）

王金柱 编著

清華大学出版社
北京

内 容 简 介

本书基于 ECMAScript 6 脚本语言规范，着重讲解将基本知识点与实际代码应用相结合，用大量易懂的、具有代表性的实例帮助读者快速学习 ECMAScript 开发。

全书共分为 16 章，内容从 ECMAScript 的基础知识到技术难点，循序渐进地呈现给读者，让读者有一个学习编程语言从易到难、由简至繁的体验过程。书中包括 ECMAScript 的发展历史、语法基础、表达式、函数、ECMAScript 对象、类、继承、ES7 与 ES8 版本新特性等方面的知识介绍。

本书是学习 ECMAScript 技术非常好的图书，相信丰富的内容和大量的实例代码能够成为读者必要的案头参考工具，成为 Web 前端开发学习者的首选。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目（CIP）数据

ECMAScript 从零开始学：视频教学版 / 王金柱编著. —北京：清华大学出版社，2018
（Web 前端技术丛书）
ISBN 978-7-302-51081-9

Ⅰ. ①E… Ⅱ. ①王… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字（2018）第 195641 号

责任编辑： 夏毓彦
封面设计： 王　翔
责任校对： 闫秀华
责任印制： 丛怀宇

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn，http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者： 北京富博印刷有限公司
装 订 者： 北京市密云县京文制本装订厂
经　　销： 全国新华书店
开　　本： 190mm×260mm　　**印　　张：** 20　　**字　　数：** 512 千字
版　　次： 2018 年 10 月第 1 版　　**印　　次：** 2018 年 10 月第 1 次印刷
定　　价： 59.00 元

产品编号：079805-01

前　言

读懂本书

学习编程主要是兴趣

学习编程是一件很枯燥的事，相信只有强烈的愿望才有坚持下去的动力。编程语言有非常多的知识点需要掌握，为了帮助读者快速入门加深理解，笔者在书中演示了大量、有趣的代码实例，期望读者能够尽快喜欢上 ECMAScript 脚本语言。

基础知识点与应用趋势

本书重点介绍了 ECMAScript 原生语法基础及其应用，特别增加了 ECMAScript 6 版本新特性的内容，对于初学者需要注意的方方面面均有特别提示，以期帮助读者尽量少走弯路。

本书真的适合你吗

本书的基础知识可以帮助读者快速踏入 ECMAScript 领域之门，可以随心所欲地把这些知识应用于实践开发。ECMAScript 6 版本新特性部分可以让读者学习到脚本语言的新技术切入点，为设计人员打开一扇全新的窗户。

本书完全是从一个新手的视角出发讲解 ECMAScript 的技术和应用，涵盖 ES6、ES7、ES8 版本的不同特性。作者遵循读者的学习心理，循序渐进、由浅入深地介绍各门类、相互关联的知识。这是一本实例书，也是一本入门引导书，是想教会你编写代码，而不是教会你语法。

本书涉及的主要软件或工具

- Windows
- Mozilla Firefox
- Notepad
- EditPlus
- Sublime Text
- WebStorm

本书涉及的技术或框架

- CSS3
- HTTP
- RegExp
- MIME
- ECMAScript 6
- DHTML
- ECMAScript
- HTML 5
- 延迟加载

本书特点

（1）本书不是强调纯粹的理论知识，也不是高深技术研讨，完全是从基础知识讲解入手，用简单的、典型的示例引申出核心知识，最后指引了通往“高精尖”进一步深入学习的道路。

（2）本书全面介绍 ECMAScript 脚本语言所涉及的前端领域、后端应用范围，能够综合性地领略到这门语言的全貌，在学习的过程中不会迷失方向。

（3）本书注重知识难点探究、技术实践结合应用场景效果，使之能激发读者的阅读兴趣且能够为读者提供编程参考。

（4）本书代码遵循重构原理，避免代码污染，真心希望读者能写出优秀的、简洁的、可维护的代码。

源代码与教学视频



本书配套的源代码与教学视频可以通过扫描右边的二维码获取下载链接，或者发邮件至 booksaga@163.com，邮件主题为“ECMAScript 从零开始学”。如果有问题或建议，也可以发至该邮箱。

读者对象

- Web 前端设计的初学者
- 从事前端开发的人员
- 喜欢或从事网页设计对前端感兴趣的人员
- 想拓展前端知识面的读者
- ECMAScript 爱好者
- JavaScript 开发人员

本书由王金柱主编，其他参与创作的还有张婷、谢志强、李一鸣、王晓华、杨旺功、陈明红、林龙、王小辉、薛燚、罗从良、陈素清、王启明，在此表示感谢。

编　者

2018 年 8 月

目　　录

第 1 章　ECMAScript 基础 1

1.1　认识 ECMAScript 1

1.1.1　ECMAScript 的基本概念 1

1.1.2　JavaScript 与 ECMAScript 的发展历史 2

1.1.3　ECMAScript 版本更迭 2

1.1.4　JavaScript 与 ECMAScript 异同 3

1.1.5　ECMAScript 脚本语言的特点 4

1.1.6　JavaScript 代码解释器 4

1.2　在网页中使用 ECMAScript 脚本语言 6

1.2.1　<script>标签 6

1.2.2　嵌入式 ECMAScript 脚本 6

1.2.3　引入外部 ECMAScript 脚本文件 8

1.3　ECMAScript 脚本运行机制 9

1.4　ECMAScript 脚本代码的开发与调试 14

1.4.1　第一步：使用 WebStorm 集成开发平台创建项目、页面文件 14

1.4.2　第二步：使用 WebStorm 集成开发平台创建脚本文件 16

1.4.3　第三步：使用 Firefox 浏览器运行 HTML 页面和调试脚本代码 17

1.5　本章小结 19

第 2 章　ECMAScript 语法 20

2.1　ECMAScript 语法规范 20

2.1.1　ECMAScript 语句 20

2.1.2　ECMAScript 大小写字母敏感 21

2.1.3　ECMAScript 代码空格 21

2.1.4　ECMAScript 代码强制换行 21

2.1.5　ECMAScript 代码注释 21

2.1.6　ECMAScript 代码块 22

2.2　ECMAScript 变量 22

2.2.1　弱类型的 ECMAScript 变量 22

2.2.2　声明 ECMAScript 变量 23

2.2.3　ECMAScript 变量命名习惯 24

2.2.4 动态改变 ECMAScript 变量类型 24
2.2.5 未声明的 ECMAScript 变量 25
2.2.6 严格模式 26
2.3 let 关键字 27
2.3.1 变量作用域 27
2.3.2 变量提升 28
2.3.3 全局变量、局部变量和块级作用域 29
2.3.4 let 关键字的简单示例 31
2.3.5 let 关键字使用规则 32
2.3.6 let 关键字应用 33
2.4 const 关键字 35
2.5 ECMAScript 关键字和保留字 38
2.6 本章小结 39

第 3 章 值与类型 40

3.1 ECMAScript 原始值与引用值 40
3.1.1 ECMAScript 原始值与引用值 40
3.1.2 ECMAScript 原始类型概述 41
3.2 Undefined 原始类型 41
3.3 Null 原始类型 44
3.4 Boolean 原始类型 46
3.5 Number 原始类型 48
3.5.1 Number 原始类型介绍 48
3.5.2 十进制 Number 原始类型 48
3.5.3 二进制 Number 原始类型 49
3.5.4 八进制 Number 原始类型 50
3.5.5 十六进制 Number 原始类型 51
3.5.6 浮点数 Number 原始类型 53
3.5.7 Number 原始类型科学计数法 54
3.6 Number 特殊值及方法 55
3.6.1 Number 最大值与最小值 55
3.6.2 Number 无穷大 56
3.6.3 非数值 NaN 58
3.6.4 Number 安全整数值 59
3.6.5 Number.EPSILON 61
3.7 String 原始类型 62
3.7.1 String 原始类型介绍 62
3.7.2 定义 String 原始类型 62
3.7.3 字符串连接 63

3.7.4　特殊字符串 64
3.7.5　获取字符串长度 65
3.7.6　字符的 Unicode 编码表示 66
3.8　本章小结 70

第 4 章　类型转换 71

4.1　转换为字符串 71
4.1.1　toString()函数方法的语法格式 71
4.1.2　使用默认 toString()函数方法 71
4.1.3　Number 类型数值转换为字符串 73
4.1.4　使用带参数的 toString()函数方法 74
4.2　转换为数值 75
4.2.1　parseInt()函数方法的语法格式 75
4.2.2　转换为整数数值 76
4.2.3　转换指定基数的整数数值 77
4.2.4　parseFloat()函数方法的语法格式 78
4.2.5　转换为浮点数 79
4.3　强制类型转换 81
4.3.1　强制类型转换基础 81
4.3.2　强制转换为 Number 类型 81
4.3.3　强制转换为 Boolean 类型 83
4.3.4　强制转换为 String 类型 84
4.4　本章小结 86

第 5 章　解构 87

5.1　ECMAScript 变量赋值机制 87
5.1.1　变量赋值机制介绍 87
5.1.2　变量赋值机制相关原理 88
5.1.3　关于变量的解构赋值 88
5.2　ECMAScript 数组解构赋值 89
5.2.1　数组解构赋值的基本方式 89
5.2.2　数组解构赋值的嵌套方式 90
5.2.3　含有空位的数组解构赋值 91
5.2.4　使用省略号的数组解构赋值 92
5.2.5　未定义的数组解构赋值 94
5.2.6　无效的数组解构赋值 95
5.2.7　使用默认值的数组解构赋值 97
5.2.8　默认值为变量的数组解构赋值 98
5.2.9　默认值为表达式的数组解构赋值 100

5.3 ECMAScript 对象解构赋值 101
5.3.1 对象解构赋值的基本方式 101
5.3.2 不按次序的对象解构赋值 102
5.3.3 对象解构赋值方式的扩展 103
5.4 ECMAScript 字符串解构赋值 104
5.5 ECMAScript 数值解构赋值 105
5.6 ECMAScript 解构赋值的应用 106
5.6.1 交换变量的值 106
5.6.2 函数返回多个值 109
5.6.3 定义函数参数 110
5.7 本章小结 112

第 6 章 运算符与表达式 113

6.1 ECMAScript 加性运算符及表达式 113
6.1.1 概述 113
6.1.2 加法运算符及表达式 113
6.1.3 减法运算符及表达式 116
6.2 ECMAScript 乘性运算符及表达式 118
6.2.1 乘性运算符与表达式概述 118
6.2.2 乘法运算符及表达式 118
6.2.3 除法运算符及表达式 120
6.2.4 取模运算符及表达式 121
6.3 ECMAScript 一元运算符及表达式 123
6.3.1 一元运算符与表达式概述 123
6.3.2 new 和 delete 运算符及表达式 123
6.3.3 void 运算符及表达式 125
6.3.4 前增量与前减量运算符及表达式 127
6.3.5 后增量与后减量运算符及表达式 128
6.3.6 一元加法与一元减法运算符及表达式 129
6.4 ECMAScript 关系运算符及表达式 131
6.4.1 关系运算符与表达式概述 131
6.4.2 数值关系运算符表达式 132
6.4.3 字符串关系运算符表达式 132
6.4.4 数值与字符串关系运算符表达式 133
6.5 ECMAScript 等性运算符及表达式 134
6.5.1 等性运算符与表达式概述 135
6.5.2 等号与不等号运算符表达式 135
6.5.3 严格相等与非严格相等运算符表达式 137
6.6 ECMAScript 位运算符及表达式 138

6.6.1 位运算符与表达式概述 139
6.6.2 整数编码介绍 139
6.6.3 NOT 位运算符及表达式 141
6.6.4 AND 位运算符及表达式 142
6.6.5 OR 位运算符及表达式 143
6.6.6 XOR 位运算符及表达式 144
6.6.7 左移运算符及表达式 145
6.6.8 保留符号位的右移运算符及表达式 146
6.6.9 无符号位的右移运算符及表达式 148
6.7 ECMAScript 逻辑运算符及表达式 149
6.7.1 逻辑运算符与表达式概述 149
6.7.2 ToBoolean 逻辑值转换操作 150
6.7.3 AND 运算符及表达式 151
6.7.4 OR 运算符及表达式 152
6.7.5 NOT 运算符及表达式 154
6.8 ECMAScript 赋值运算符及表达式 156
6.9 ECMAScript 条件运算符及表达式 157
6.10 本章小结 159

第 7 章 流程控制语句 160

7.1 if 条件语句 160
7.1.1 if 语句 160
7.1.2 if…else…语句 161
7.1.3 if…else if…else…语句 162
7.2 switch 条件语句 164
7.3 循环迭代语句 166
7.3.1 for 语句 166
7.3.2 for…in…语句 168
7.3.3 while 语句 170
7.3.4 do…while 语句 170
7.4 循环中断语句 171
7.4.1 break 语句 172
7.4.2 continue 语句 172
7.4.3 break 语句与标签语句配合使用 173
7.4.4 continue 语句与标签语句配合使用 175
7.5 ECMAScript 6 新特新——for of 迭代循环 177
7.5.1 迭代数组 177
7.5.2 迭代字符串 177
7.5.3 for of 循环迭代原理 178

7.6 本章小结......178
第 8 章 函数......179
8.1 ECMAScript 函数基础......179
8.2 ECMAScript 函数声明、定义与调用......180
8.2.1 传统方式定义 ECMAScript 函数......180
8.2.2 ECMAScript 函数表达式方式......181
8.2.3 Function 构造方式定义 ECMAScript 函数......184
8.3 ECMAScript 函数返回值......185
8.4 arguments 对象......187
8.5 Function 对象......190
8.5.1 Function 对象实现函数指针......190
8.5.2 Function 对象属性......192
8.5.3 Function 对象方法......193
8.6 本章小结......194
第 9 章 系统函数......195
9.1 ECMAScript 常规函数......195
9.1.1 常规函数介绍......195
9.1.2 警告对话框和确认对话框......195
9.1.3 parseInt()函数......197
9.1.4 isNaN()函数......198
9.1.5 eval()函数......199
9.2 ECMAScript 字符串函数......200
9.3 ECMAScript 数学函数......203
9.4 ECMAScript 数组函数......204
9.4.1 数组函数介绍......204
9.4.2 join 函数......205
9.4.3 reverse 函数......205
9.4.4 sort 函数......206
9.4.5 from 函数......207
9.5 ECMAScript 日期函数......209
9.6 本章小结......210
第 10 章 函数扩展......211
10.1 ECMAScript 函数参数扩展......211
10.1.1 可变参数......211
10.1.2 rest 参数......213
10.1.3 参数默认值......214

10.1.4 省略参数默认值的正确方式 216
10.2 length 属性扩展 218
10.2.1 参数默认值方式下的 length 属性 218
10.2.2 rest 参数方式下的 length 属性 219
10.2.3 参数默认值不同位置下的 length 属性 220
10.3 name 属性扩展 220
10.4 箭头函数 221
10.4.1 箭头函数的基本形式 221
10.4.2 箭头函数的参数 222
10.4.3 箭头函数的函数体 224
10.5 箭头函数扩展应用 224
10.5.1 箭头函数计算工具 225
10.5.2 箭头函数与解构赋值 226
10.5.3 箭头函数与回调函数 227
10.5.4 箭头函数与链式函数 228
10.6 本章小结 229

第 11 章 ECMAScript 对象 230

11.1 ECMAScript 对象 230
11.1.1 什么是 ECMAScript 对象 230
11.1.2 ECMAScript 对象构成 230
11.1.3 ECMAScript 对象实例 231
11.2 创建 ECMAScript 对象 231
11.3 ECMAScript 对象初始化 231
11.4 ECMAScript 对象销毁 233
11.5 ECMAScript 对象绑定方式 235
11.6 本章小结 235

第 12 章 对象类型 236

12.1 ECMAScript 对象概述 236
12.2 Object 对象 237
12.3 String 对象 238
12.4 Array 对象 239
12.4.1 Array 对象初始化 239
12.4.2 Array 对象连接操作 241
12.4.3 Array 对象模拟堆栈 242
12.5 Number 对象 244
12.6 Boolean 对象 245
12.7 Date 对象 246

12.7.1 Date 对象基础 247
12.7.2 Date 对象应用（一） 248
12.7.3 Date 对象应用（二） 249
12.8 本章小结 251

第 13 章 对象新特性 252

13.1 对象属性的简洁表示法 252
13.2 Symbol 数据类型 254
13.2.1 定义 Symbol 对象 254
13.2.2 Symbol 对象的唯一性 254
13.2.3 Symbol 定义属性名 255
13.3 Set 数据类型 256
13.3.1 定义和遍历 Set 数据类型 256
13.3.2 判断 Set 集合中的值 257
13.3.3 删除和清空 Set 集合 258
13.4 Map 数据类型 259
13.4.1 定义 Map 数据类型和基本存取操作 259
13.4.2 判断 Map 集合中的值 260
13.4.3 删除和清空 Map 集合 261
13.5 本章小结 262

第 14 章 正则表达式 263

14.1 正则表达式基础 263
14.1.1 什么是正则表达式 263
14.1.2 RegExp 对象语法 263
14.1.3 RegExp 对象模式 264
14.2 RegExp 对象方法 267
14.2.1 test 方法 267
14.2.2 exec 方法 268
14.2.3 compile 方法 269
14.3 RegExp 对象修饰符标记 270
14.3.1 “g”修饰符标记 271
14.3.2 “i”修饰符标记 272
14.3.3 “g”和“i”修饰符标记组合 273
14.4 本章小结 274

第 15 章 面向对象编程 275

15.1 面向对象基础 275
15.1.1 什么是“面向对象” 275

15.1.2 面向对象的特点 276
15.1.3 面向对象的专业术语 276
15.2 ECMAScript 对象作用域 277
15.2.1 对象作用域 277
15.2.2 this 关键字 277
15.3 创建 ECMAScript 类与对象 278
15.3.1 工厂模式创建类与对象 278
15.3.2 封装的工厂模式创建类与对象 279
15.3.3 带参数的工厂模式创建类与对象 281
15.3.4 工厂模式的最大局限 282
15.3.5 构造函数方式创建类与对象 283
15.3.6 原型方式创建类与对象 284
15.3.7 结合构造函数方式与原型方式创建类和对象 285
15.4 原型 Prototype 应用 286
15.4.1 定义新方法 286
15.4.2 重定义已有方法 287
15.4.3 实现继承机制 288
15.5 ECMAScript 6 面向对象新特性 291
15.5.1 通过“class”定义类 291
15.5.2 通过“extends”继承类 292
15.5.3 类的 setter 和 getter 方法 293
15.6 本章小结 294

第 16 章 ECMAScript 7 & 8 版本新特性 295

16.1 ECMAScript 7 & 8 版本的新特性 295
16.2 ECMAScript 7（2016）版本的新特性 295
16.2.1 Array.prototype.includes()方法 296
16.2.2 指数操作符 297
16.3 ECMAScript 8（2017）版本的新特性 298
16.3.1 字符串填充（String Padding） 298
16.3.2 对象遍历 300
16.3.3 异步函数（Async Function） 301
16.4 本章小结 305

第 1 章

ECMAScript 基础

开篇之章，将向读者介绍 ECMAScript 的基本概念、发展历史、版本更迭、组成部分、基本特性及使用方法这些基础内容。另外，ECMAScript 源自于 JavaScript 脚本语言，二者之间的异同点也是本章的重点内容。总之，ECMAScript 语言易学易用、技术领先、功能强大等特点是 Web 前端设计人员学习使用 JavaScript 脚本语言的重要基础（ECMAScript 是标准，JavaScript 是它的实现）。

1.1　认识 ECMAScript

首先简单介绍一下 ECMAScript 脚本语言的基本概念、发展历史、版本更迭和组成部分等内容。

1.1.1　ECMAScript 的基本概念

ECMAScript 是由 Ecma 国际（Ecma International）通过 ECMA-262 标准化规范而设计的脚本程序设计语言。这个 Ecma 国际前身即欧洲计算机制造商协会（European Computer Manufacturers Association，ECMA）。也就是说，ECMAScript 是一种由标准组织推出的程序设计语言。

同时，在互联网设计开发中被广泛使用的 JavaScript 脚本语言，实际上是基于 ECMA-262 标准规范而设计实现的。因此，可以认为 ECMAScript 与 JavaScript 本质上是一种语言，但也要清楚地认识二者之间的区别。

不过，如果读者想真正了解 ECMAScript 与 JavaScript 之间的异同，首先要了解一下二者的发展历史。

1.1.2 JavaScript 与 ECMAScript 的发展历史

去查阅一下 JavaScript 与 ECMAScript 二者的发展历史，就会了解很多有趣的历史故事，同时也将体会到 JavaScript 脚本语言能够走到今天的艰辛。

JavaScript 脚本语言的发展历史可以用一波三折、精彩纷呈来形容。JavaScript 最初是由著名的 Netscape 公司（网景公司）的 Brendan Eich 于 1995 年设计提出的，当然最初也是在 Netscape 浏览器上实现的。设计 JavaScript 这种脚本语言的目的是为了增强浏览器功能、提高用户体验。

其实，当时 JavaScript 最初的命名是 LiveScript，后来由于 Netscape 公司与 Sun 公司进行合作才改名为 JavaScript。至于改名为 JavaScript 的原因，相信大多数读者猜到了，主要是源于 Sun 公司非常著名的软件产品—— Java 语言。设计者的初衷是想让 JavaScript 也能够像 Java 那样流行。因此，今天的 JavaScript 在语法和命名规范上或多或少都有 Java 语言的影子，二者确实有着千丝万缕的渊源。但请读者一定要注意，JavaScript 与 Java 本质上是完全不同的两类程序设计语言。

JavaScript 脚本语言在发展初期并没有确立所谓的统一标准，但因为其在 Netscape 浏览器上的惊艳表现，随后其他软件生产商也陆续推出了自己的产品。比较有名的有 Miscrosoft 公司的 Jscript 语言和 CEnvi 的 ScriptEase 语言，与 JavaScript 语言一样都可以在浏览器中运行。尤其是 Jscript 语言，其与 Internet Explorer 浏览器可谓是相得益彰，在与 Netscape 公司浏览器的竞争中也是后来居上。因此，早期的 JavaScript 脚本语言并没有统一的标准，完全是各大浏览器软件厂商在“各自为政”。

为了避免各大浏览器软件厂商在各自的 JavaScript 标准上越走越远，1997 年在 ECMA 的提议协调下，由 Netscape、Sun、Miscrosoft 和 Borland 等公司组成了工作组，最终确定了统一的脚本语言标准规范—— ECMA-262，而 ECMA-262 标准规范也就是 ECMAScript。

目前，ECMA-262 标准规范就是事实上的脚本语言设计标准，各大浏览器软件厂商在各自的浏览器软件产品上实现脚本功能时都必须遵循 ECMA-262 规范，这样就可以保证良好的兼容性了。当然，各大浏览器软件厂商在实现一些功能特效时又各有各的特点，这也是脚本语言跨平台设计时需要设计人员注意的。

1.1.3 ECMAScript 版本更迭

ECMA-262 标准规范发布至今，ECMAScript 主要有 8 个版本，具体如下：

（1）ECMAScript 1

1997 年 6 月发布了 ECMAScript 1.0 版，即最初在 ECMA 的提议协调下，由 Netscape、Sun、Miscrosoft 和 Borland 等公司组成的工作组共同研究发布的。

（2）ECMAScript 2

1998 年 6 月发布了 ECMAScript 2.0 版，并进行了格式修正，使其形式与 ISO/IEC16262 国际标准保持一致。

（3）ECMAScript 3

1999 年 12 月 ECMAScript 3.0 版发布，实现了正则表达式功能，提供了更好的文字链处理方式和新的控制指令，对异常处理和错误定义更加明确等。这个版本成为 JavaScript 脚本语言的通行

标准，得到前端设计人员的广泛支持。

（4）ECMAScript 4

ECMAScript 4 是很复杂的一个版本。2007 年 10 月 ECMAScript 4.0 版草案发布出来后，由于升级改进的程度过大，各方发生了严重分歧。大型的软件公司（如 Microsoft、Google 和 Yahoo 等）均反对进行大幅度升级改进，而以 Brendan Eich（JavaScript 最初设计者）为首的 Mozilla 公司则坚持主张按照草案执行。最终，ECMA 决定中止 ECMAScript 4.0 版本的开发，对草案中涉及现有功能改善的一小部分内容发布为 ECMAScript 3.1 版本，对于草案中的大部分内容留在后续版本开发中实现。

另外，2004 年 6 月 ECMA（欧洲计算机制造商协会）发表了 ECMA-357 标准，这是 ECMAScript 的一个扩展，因此也被称为 E4X（ECMAScript for XML）。

（5）ECMAScript 5

2009 年 12 月，ECMAScript 5.0 版正式发布。2011 年 6 月，ECMAScript 5.1 版发布，并且成为 ISO 国际标准（ISO/IEC 16262:2011）。

（6）ECMAScript 6（ECMAScript 2015）

2013 年 3 月，ECMAScript 6 草案定稿，将不再添加新功能。2015 年 6 月 17 日，著名的 ECMAScript 6 正式版发布，即 ECMAScript 2015。ECMAScript 6 是继 ECMAScript 5 之后的一次重要改进，增添了许多重要特性（如模块和类、Maps、Sets、Promises、Generators 等）。同时，ECMAScript 6 保证了较大程度地兼容以前的版本，所有旧代码都可以正常运行。当然，许多开发者们抱怨了多年的老问题也依然存在。

ECMA 的第 39 号技术专家委员会（Technical Committee 39，TC39）负责制定 ECMAScript 标准，成员包括 Microsoft、Mozilla、Google 等大公司。TC39 的职责就是总体保证 ECMAScript 新版本的基本兼容性，在较大的语法修正及新功能特性增加方面兼顾老版本的语言支持。另外，TC39 规定了从 ECMAScript 6 开始，新版本将按照年份的格式（如 ECMAScript 2015）进行发布。

（7）ECMAScript 7（ECMAScript 2016）

ECMAScript 2016 标准规范定义了一个重要的制定原则，即成文标准必须从事实标准中诞生，也就是要先实现再制定标准。同时，如果想要进入标准草案，就必须有两个以上的 JavaScript 引擎实现的支持。

（8）ECMAScript 8（ECMAScript 2017）

ECMAScript 2017 标准规范中定义了许多新特性，包括字符串填充、对象值遍历、对象的属性描述符获取、函数参数列表、调用中的尾部逗号、异步函数、共享内存与原子操作等。

1.1.4　JavaScript 与 ECMAScript 异同

虽然在绝大多数情况下，设计人员是不区分 JavaScript 与 ECMAScript 这两个概念的。但是，从严格意义上来说，JavaScript 与 ECMAScript 还是有所区别的。图 1.1 是对 JavaScript 与 ECMAScript 之间关系的一个简单概括。

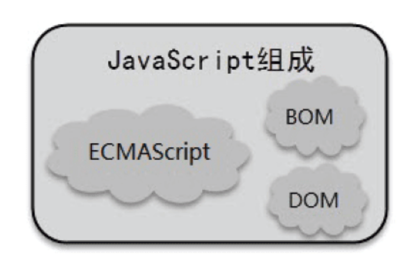


图 1.1　JavaScript 与 ECMAScript

从图 1.1 中可以看到，整体 JavaScript 脚本语言包含三个组成部分：ECMAScript 标准规范、文档对象模型（DOM）和浏览器对象模型（BOM）。关于这三个组成部分的描述如下：

- 在设计实现 JavaScript 脚本语言的语法和基本对象这些核心内容时所遵循的就是 ECMAScript 标准规范。
- 文档对象模型（DOM）是用于描述 JavaScript 脚本语言处理网页内容的方法和接口。
- 浏览器对象模型（BOM）是用于描述 JavaScript 脚本语言与浏览器进行交互的方法和接口。

因此，从严格意义上来说，ECMAScript 是 JavaScript 脚本语言组成中的一部分，也是最核心的部分。JavaScript 脚本语言在遵循 ECMAScript 标准规范（ECMA-262）设计的同时，还设计实现了与浏览器接口交互等功能。

1.1.5　ECMAScript 脚本语言的特点

ECMAScript 是一种应用于 Web 程序开发的脚本语言，主要用来增强网页的动态功能，提高用户的交互体验。ECMAScript 脚本语言的主要特点如下：

- ECMAScript 是一种解释性脚本语言（直译式），需要借助浏览器和解释器（JavaScript 脚本引擎）执行。
- ECMAScript 脚本语言通常是嵌入在 HTML 网页代码中实现交互功能的。
- ECMAScript 脚本语言具有很友好的跨平台特性（如 Windows、Linux、Mac、Android、iOS 等平台），同样也具有跨浏览器特性。
- ECMAScript 脚本语言具有面向对象程序设计功能。
- 基于 ECMAScript 脚本语言开发的前端框架十分丰富，功能也十分强大。

ECMAScript 脚本语言与其他编程语言一样，支持基本数据类型、表达式、算术运算符及基本程序框架。ECMAScript 脚本语言提供了 4 种基本的数据类型和两种特殊数据类型用来处理数据和文字，而 ECMAScript 表达式则可以完成比较复杂的信息处理。

1.1.6　JavaScript 代码解释器

本质上 JavaScript 是一种直译式的脚本语言，是内置支持动态类型、弱类型、基于原型的编程语言。JavaScript 作为直译式的脚本语言，必然需要代码解释器来执行脚本程序。

所谓代码解释器，其实就是在执行程序时负责将代码解释成机器语言，然后交由计算机操作系统运行的工具。所以，代码解释器本质上也是一种计算机程序，类似于 Java 虚拟机的概念。只不过 JavaScript 代码解释器是负责运行脚本代码的程序，在操作系统中处于较高的级别。

代码解释器对于诸如 JavaScript 此类的直译式脚本语言是极为重要的，算得上是最核心的组成部分。因此，设计人员习惯上也将 JavaScript 代码解释器称为"JavaScript 引擎"。JavaScript 引擎就是运行 JavaScript 脚本代码的核心部分。

目前，主流浏览器厂商的做法是将 JavaScript 代码解释器全部内置于浏览器内部。虽然，各大浏览器厂商在具体实现功能上各有特点，但必须遵循 ECMA（欧洲计算机制造商协会）的 EMCA-262 标准规范进行开发，这样才能保证最大程度的代码兼容性。

目前，主流的 JavaScript 代码解释器均是依托于主流浏览器而存在的。这也契合当前 EMCA-262 标准规范的制定原则，只有功能完全实现并广泛应用了的才可能成为 EMCA-262 标准规范。

下面就列举一些当前主流的 JavaScript 代码解释器。

- Google 的 Chrome 浏览器

Chrome 浏览器使用的是著名的 V8 引擎，全部开放源代码，内置于 Chrome 浏览器。V8 引擎是目前影响力、性能都很强大的 JavaScript 代码解释器，许多主流浏览器均借鉴了其技术。

- Mozilla 的 FireFox 浏览器

FireFox 浏览器所使用的 JavaScript 代码解释器经历了很复杂的历史演变过程。早期的 Mozilla Firefox 1.0～3.0 版本使用的是 SpiderMonkey，是由 Brendan Eich 在 Netscape Communications 时期编写的，也是业界的第一款 JavaScript 引擎。

TraceMonkey 是基于实时编译的 JavaScript 代码解释器，其中部分代码取自 Tamarin 引擎技术，主要用于 Mozilla Firefox 3.5～3.6 版本。

JaegerMonkey 是结合了追踪和组合码技术的 JavaScript 代码解释器，程序性能大幅提高。JaegerMonkey 部分借鉴了 Google V8、JavaScriptCore、WebKit 等技术，主要用于 Mozilla Firefox 4.0 以上版本。

- 微软的 IE 系列和 Edge 浏览器

早期的 Internet Explorer（IE3～IE8）版本使用的是微软自己的 JScript 代码解释器，在 IE9 版本后使用的是名称为 Chakra（查克拉）的 JavaScript 代码解释器。随着 Windows 10 一同发布的 Microsoft Edge 浏览器沿用了 Chakra 引擎，做了很多改进。

- Opera 浏览器

Opera 浏览器使用的是 JavaScript 引擎，也很复杂。Linear A 引擎用于 Opera 4.0～6.1 版本，Linear B 引擎用于 Opera 7.0～9.2 版本，Futhark 引擎用于 Opera 9.5～10.2 版本，而 Carakan 引擎是由 Opera 软件公司自己编写的，在 Opera10.50 版本以后开始使用。

- Apple 的 Safari 浏览器

目前的 Safari 浏览器使用的是 Nitro 引擎，也就是原来的 SquirrelFish 系列引擎。

- Tamarin 引擎

前文提到的 Tamarin 引擎是由 Adobe Labs 编写的，只有 Flash Player 9 在使用。

1.2 在网页中使用 ECMAScript 脚本语言

本节我们介绍如何在 HTML 网页中使用 ECMAScript 脚本语言，具体就是使用<script>标签编写 ECMAScript 脚本代码的方法。

1.2.1 <script>标签

如果想在HTML网页中使用ECMAScript脚本语言，那么一定会用到<script>标签。通过<script>标签既可以编写嵌入式脚本代码，也可以引入外部脚本文件。同时，<script>标签作为一种标记符号，均需要成对（开始标记<script>和结束标记</script>）使用。

下面是嵌入式脚本代码的具体示例：

```
<script>
    // TODO: 脚本代码
    …
</script>
```

以上写法是基于 HTML5 版本网页的，而对于 HTML 4.01 版本的网页，则一定要使用下面的写法：

```
<script type="text/javascript">
    // TODO: 脚本代码
    …
</script>
```

如上代码所示，需要在<script>标签中加入“type="text/javascript"”的属性描述，才能让浏览器正确识别 HTML 网页。

下面是引入外部脚本文件的具体代码示例：

```
<script src="ECMAScript.js"></script>
```

以上是常规写法，还可以使用下面的简化写法：

```
<script src="ECMAScript.js" />
```

如上代码所示，可以省略结束标记</script>，但需要同时在开始标记<script>的最后加上符号（/）表示标记结束。

1.2.2 嵌入式 ECMAScript 脚本

嵌入式 ECMAScript 脚本是在网页中使用脚本代码比较简单直接的方法。该方式的优点就是可

以将脚本代码插入 HTML 网页中的任何位置，只需要使用<script>标签编写 ECMAScript 脚本代码即可。

下面是一个在 HTML 网页中使用嵌入式 ECMAScript 脚本代码的示例（详见源代码 ch01 目录中的 ch01-es-embed.html 文件）。

【代码 1-1】

```
01  <!doctype html>
02  <html lang="en">
03  <head>
04   <!-- 添加文档头部内容 -->
05   <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
06   <meta http-equiv="Content-Language" content="zh-cn" />
07   <link rel="stylesheet" type="text/css" href="css/style.css">
08   <script>
09        alert("ECMAScript - 嵌入脚本代码");
10   </script>
11   <title>ECMAScript in 15-days</title>
12  </head>
13  <body>
14   <!-- 添加文档主体内容 -->
15   <header>
16        <nav>ECMAScript - 嵌入脚本代码</nav>
17   </header>
18   <hr>
19   <!-- 添加文档主体内容 -->
20  </body>
21  </html>
```

关于【代码 1-1】的分析如下：

第 08～10 行代码通过<script>标签定义了一段嵌入式 ECMAScript 脚本代码，第 09 行的脚本代码通过“alert()”函数定义了一个弹出式警告提示框。

下面通过 FireFox 浏览器测试一下【代码 1-1】所定义的 HTML 页面，查看页面中嵌入式 ECMAScript 脚本代码的执行效果，如图 1.2 所示。

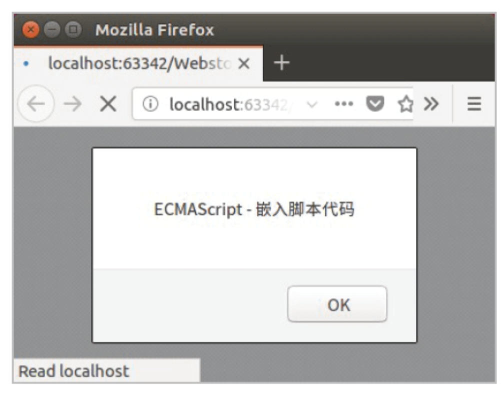


图 1.2　嵌入式 ECMAScript 脚本代码

如图 1.2 所示，【代码 1-1】中第 09 行脚本代码定义的弹出式警告提示框在页面加载过程中成功显示出来了。注意，到此时页面应该还没有加载完成，关于这一点我们在后文中会给出原因分析。

1.2.3 引入外部 ECMAScript 脚本文件

在 HTML 网页中引入外部 ECMAScript 脚本文件是另一种使用脚本语言的方法，这种方法非常适用于需要使用大量 ECMAScript 脚本代码的情况。一般来说，这种方法称为外链式 ECMAScript 脚本。

外链式 ECMAScript 脚本的基本使用方法如下：

```
<script src="xxx.js"></script>
```

这里，“src”属性用于定义外部 ECMAScript 文件的路径地址。其中，路径既可以是绝对路径，也可以是相对路径，需要根据具体情况来定。

下面将【代码 1-1】稍加修改，按照外链式 ECMAScript 脚本进行设计，具体代码如下（详见源代码 ch01 目录中的 ch01-es-linking.html 文件）。

【代码 1-2】

```
01  <!doctype html>
02  <html lang="en">
03  <head>
04   <!-- 添加文档头部内容 -->
05   <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
06   <meta http-equiv="Content-Language" content="zh-cn" />
07   <link rel="stylesheet" type="text/css" href="css/style.css">
08   <script type="text/javascript" src="js/ch01-es-linking.js"></script>
09   <title>ECMAScript in 15-days</title>
10  </head>
11  <body>
12   <!-- 添加文档主体内容 -->
13   <header>
14        <nav>ECMAScript - 外链式脚本文件</nav>
15   </header>
16   <hr>
17   <!-- 添加文档主体内容 -->
18  </body>
19  </html>
```

关于【代码 1-2】的分析如下：

第 08 行代码通过<script>标签定义了外链式 ECMAScript 脚本文件，其中“src”属性定义了外部脚本的相对路径地址（"js/ch01-es-linking.js"）。

关于上面引入的外部脚本文件的具体代码如下（详见源代码 ch01 目录中的 js/ch01-es-linking.js 文件）。

【代码 1-3】

```
01  alert("ECMAScript - 外链式脚本文件");
```

这行脚本代码通过“alert()”函数定义了一个弹出式警告提示框。

运行【代码 1-2】使用外链式 ECMAScript 脚本文件定义的 HTML 页面，页面打开后的效果如图 1.3 所示。

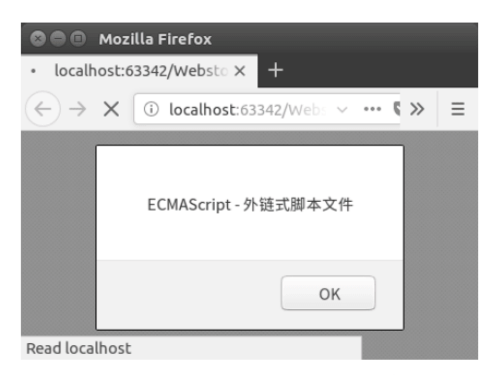


图 1.3　外链式 ECMAScript 脚本

通过图 1.3 与图 1.2 的效果对比，可以发现在 HTML 页面中无论是使用外链式脚本还是嵌入式脚本，实现的功能是完全一样的。

1.3　ECMAScript 脚本运行机制

理论上，ECMAScript 脚本代码可以定义在 HTML 网页中的任何位置。比如，可以定义在页面头部<head>标签内，也可以定义在页面主体<body>标签内任意位置，还可以定义在页面<body>标签之后。那么，ECMAScript 脚本代码定义在 HTML 页面中的不同位置有没有什么区别呢？

要想解答这个疑问，就要先理解 HTML 网页的运行机制。HTML 网页是按照页面代码定义的先后顺序自上而下依次执行的。我们在浏览网页时就会发现，页面内容其实是自上而下依次刷新出来的。因此，对于定义在 HTML 页面中的 ECMAScript 脚本代码，会随着 HTML 网页自上而下的顺序执行。同时，ECMAScript 脚本代码自身是按照中断机制执行的，也就是说 HTML 网页遇到脚本代码时会中止执行，直到脚本代码解析完成后网页才会继续运行。

ECMAScript 脚本代码的运行机制会带来一个很大的问题，那就是 ECMAScript 脚本代码定义在 HTML 网页中不同的位置会对 HTML 页面在浏览器中的显示效果产生很大的影响。

下面先看一个将 ECMAScript 脚本代码定义在页面头部<head>标签中的实例，具体代码如下（详见源代码 ch01 目录中的 ch01-es-run-in-head.html 文件）。

【代码 1-4】

```
01  <!doctype html>
02  <html lang="en">
03  <head>
04   <!-- 添加文档头部内容 -->
05   <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
06   <meta http-equiv="Content-Language" content="zh-cn" />
07   <link rel="stylesheet" type="text/css" href="css/style.css">
08   <script>
09       alert("ECMAScript 脚本代码定义在页面头部<head>标签元素内.");
10   </script>
11   <title>ECMAScript in 15-days</title>
12  </head>
```

```
13  <body>
14    <!-- 添加文档主体内容 -->
15    <header>
16          <nav>ECMAScript - 脚本代码运行机制</nav>
17    </header>
18    <hr>
19    <!-- 添加文档主体内容 -->
20    <h3>正文</h3>
21    <p>
22          ECMAScript 脚本代码定义在页面头部&lt;head&gt;标签元素内.
23    </p>
24  </body>
25  </html>
```

关于【代码 1-4】的分析如下：

第 08～10 行代码通过<script>标签定义了一段嵌入式 ECMAScript 脚本代码，第 09 行的 js 代码通过“alert()”函数定义了一个弹出式警告提示框。

第 13～24 行代码<body>标签内定义了一些页面内容，包括标题和正文文本。

运行【代码 1-4】定义的 HTML 页面查看一下 ECMAScript 脚本代码的执行效果，如图 1.4 所示。

如图 1.4 所示，第 08～10 行代码定义的警告提示框弹出来了，但页面中定义的文本内容却没有显示出来，浏览器窗口还是灰色的。这就说明了 ECMAScript 脚本代码的中断执行机制，由于本例脚本代码定义在<head>标签内，因此页面内容还没有加载完成就被运行的脚本代码中断了。

单击警告提示框中的 OK 按钮将脚本代码执行完成，效果如图 1.5 所示。



图 1.4　定义在页面头部的 ECMAScript 脚本（1）

图 1.5　定义在页面头部的 ECMAScript 脚本（2）

如图 1.5 所示，直到 ECMAScript 脚本代码执行完毕后，页面内容才会加载完成并显示在浏览器中。

下面继续看一个将 ECMAScript 脚本代码定义在页面主体<body>标签中的实例（ECMAScript 脚本是允许定义在 HTML 页面中任何位置的），具体代码如下（详见源代码 ch01 目录中的 ch01-es-run-in-body.html 文件）。

【代码 1-5】

```
01  <!doctype html>
02  <html lang="en">
```

```
03 <head>
04  <!-- 添加文档头部内容 -->
05  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
06  <meta http-equiv="Content-Language" content="zh-cn" />
07  <link rel="stylesheet" type="text/css" href="css/style.css">
08  <title>ECMAScript in 15-days</title>
09 </head>
10 <body>
11  <!-- 添加文档主体内容 -->
12  <header>
13       <nav>ECMAScript - 脚本代码运行机制</nav>
14  </header>
15  <hr>
16  <!-- 添加文档主体内容 -->
17  <script>
18       alert("ECMAScript 脚本代码定义在页面主体<body>标签元素内.");
19  </script>
20  <h3>正文</h3>
21  <p>
22       ECMAScript 脚本代码定义在页面主体&lt;body&gt;标签元素内.
23  </p>
24 </body>
25 </html>
```

关于【代码 1-5】的分析如下：

第 17～19 行代码通过<script>标签定义了一段嵌入式 ECMAScript 脚本代码，其定义位置是在页面主体<body>标签中的，第 18 行的脚本代码通过“alert()”函数定义了一个弹出式警告提示框。

在第 17～19 行脚本代码的前后分别定义了一些页面内容，包括标题和正文文本。

运行【代码 1-5】定义的 HTML 页面查看一下 ECMAScript 脚本代码的执行效果，如图 1.6 所示。

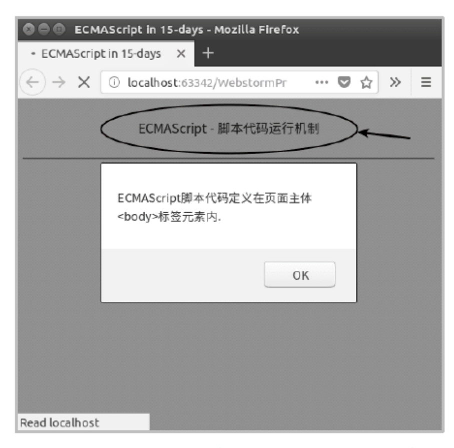


图 1.6　定义在页面主体中的 ECMAScript 脚本（1）

如图 1.6 所示，第 17～19 行代码定义的警告提示框弹出来了，同时大家注意图中箭头指向的内容：第 12～14 行代码定义的页面标题和第 15 行代码水平分割线已经显示出来了，但第 20～23 行代码定义的页面正文却没有显示出来。

这是由 ECMAScript 脚本代码的中断执行机制引起的，因为本例中脚本代码定义在第 17～19 行，正好在页面文本内容的中间。

单击警告提示框中的 OK 按钮将脚本代码执行完毕，效果如图 1.7 所示。



图 1.7　定义在页面主体中的 ECMAScript 脚本（2）

如图 1.7 所示，直到 ECMAScript 脚本代码执行完毕后，后面的页面内容才加载完成显示在浏览器中。

最后，看一个将 ECMAScript 脚本代码定义在页面主体<body>标签后的实例（之后会有总结分析），具体代码如下（详见源代码 ch01 目录中的 ch01-es-run-in-end.html 文件）。

【代码 1-6】

```
01  <!doctype html>
02  <html lang="en">
03  <head>
04   <!-- 添加文档头部内容 -->
05   <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
06   <meta http-equiv="Content-Language" content="zh-cn" />
07   <link rel="stylesheet" type="text/css" href="css/style.css">
08   <title>ECMAScript in 15-days</title>
09  </head>
10  <body>
11   <!-- 添加文档主体内容 -->
12   <header>
13        <nav>ECMAScript - 脚本代码运行机制</nav>
14   </header>
15   <hr>
16   <!-- 添加文档主体内容 -->
17   <h3>正文</h3>
18   <p>
19        ECMAScript 脚本代码定义在页面主体&lt;body&gt;标签元素后.
20   </p>
21  </body>
22  <script type="text/javascript" src="js/ch01-es-run-in-end.js"></script>
23  </html>
```

关于【代码 1-6】的分析如下：

第 10～21 行代码在页面主体<body>标签内定义了一些页面内容，包括标题和正文文本。

第 22 行代码通过<script>标签定义了外链式 ECMAScript 脚本，其中“src”属性定义了外部脚本的相对路径地址（"js/ch01-es-run-in-end.js"）；这里 ECMAScript 脚本的位置是放在<body>标签

后的，即 HTML 网页的最后。

关于上面引入的外部脚本文件的具体代码如下（详见源代码 ch01 目录中的 js/ch01-es-run-in-end.js 文件）。

【代码 1-7】

```
01  alert("ECMAScript 脚本代码定义在页面主体<body>标签元素后.");
```

关于【代码 1-7】的分析如下：

第 01 行 js 代码通过“alert()”函数定义了一个弹出式警告提示框，与前面两个实例的 JS 脚本功能类似。

运行【代码 1-6】定义的 HTML 页面查看一下 ECMAScript 脚本代码的执行效果，如图 1.8 所示。

如图 1.8 所示，第 17～19 行代码定义的警告提示框弹出来了，同时大家注意图中箭头指向的内容（部分内容被弹出框遮盖了）；第 10～21 行代码定义的页面内容已经全部显示出来了。这说明本例中引用在页面主体<body>标签后的脚本代码是在页面内容全部加载完毕后执行的，与 ECMAScript 脚本定义的位置是一致的。

单击警告提示框中的 OK 按钮将脚本代码执行完毕，效果如图 1.9 所示。

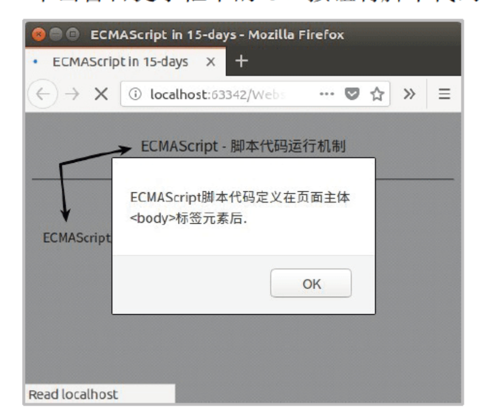


图 1.8 定义在页面最后的 ECMAScript 脚本（1）

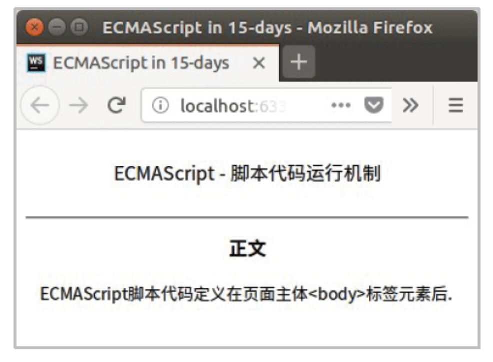


图 1.9 定义在页面最后的 ECMAScript 脚本（2）

通过以上 3 个实例，大致了解了 ECMAScript 脚本代码在 HTML 网页中定义的位置对于页面执行效果的影响。下面简单总结一下 ECMAScript 脚本代码的位置在 HTML 网页中定义的原则。

第一，尽可能地将 ECMAScript 脚本代码（包括嵌入式和外链式）放在<body>标签之后，这样在 HTML 网页内容加载时就不会因为脚本代码的中断执行机制而延迟阻塞，自然就提高了页面的显示速度。

第二，如果要实现一些页面特效，需要预先动态加载一些 ECMAScript 脚本代码，那么这些脚本代码应该放在<head>标签内或<body>标签的前面。

第三，对于需要使用 ECMAScript 脚本代码动态访问操作页面 DOM 元素的情况，要将脚本代码放在 DOM 元素定义之后（依据第一条原则，最好统一放在<body>标签之后）；否则，如果放在 DOM 元素定义之前，就会因为 DOM 元素还没有生成而出现脚本代码访问错误或无效操作的情况。

以上就是 ECMAScript 脚本代码定义位置的基本原则，当然原则绝不是一成不变的，需要根据具体情况灵活运用。

1.4 ECMAScript 脚本代码的开发与调试

本节将通过一个实例介绍 ECMAScript 脚本语言开发与调试的基本方法，包括 ECMAScript 脚本语言开发工具和调试工具的使用。

开发与调试 ECMAScript 脚本语言的方式非常灵活，可供选择的工具也非常多。我们既可以使用相对简单的轻量级代码编辑器（如 EditPlus、Sublime Text、UltraEdit 等），也可以使用功能非常强大的集成开发平台（如 Visual Studio、jetBrains WebStorm、Adobe Dreamweaver 等）编辑脚本代码。前一类属于轻量级的代码编辑器，具有简洁快速的特点，同时功能也非常健全，唯一不足的是调试代码需要自行配置的内容较多；后一类属于重量级集成开发平台，具有强大的代码管理和调试功能，对于开发大型 Web 项目是比较好的选择，如果使用得当，那么效率会成倍提高。

另外，对于 ECMAScript 脚本语言的调试，专业的做法是使用带有脚本代码调试功能的浏览器来进行。本书选用了目前比较流行的 Google Chrome、Firefox 和 Opera 这几款主流厂商的浏览器。这几款浏览器均内置有脚本代码调试功能，且调试界面、功能和方法大同小异，读者可根据个人偏好自行选择一款浏览器进行测试。

下面就通过一个具体的 ECMAScript 脚本代码案例介绍一下脚本代码开发与调试的基本步骤。

- 操作系统环境：Ubuntu 16.04 LTS 版本（基于 ECMAScript 脚本语言的平台无关性，本书的示例代码完全兼容 Windows 操作系统，请读者直接使用 Windows 操作系统）。
- 集成开发平台：WebStorm 2017.3。
- 代码编辑器：Sublime Text 3。
- 浏览器：FireFox。

1.4.1 第一步：使用 WebStorm 集成开发平台创建项目、页面文件

先打开 WebStorm 开发平台，如图 1.10 所示。

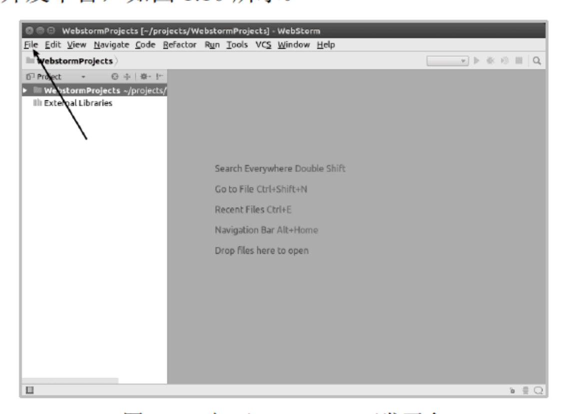


图 1.10 打开 WebStorm 开发平台

如图 1.10 中箭头所指，在文件（File）菜单中选择“New | Projects”选项，如图 1.11 所示。

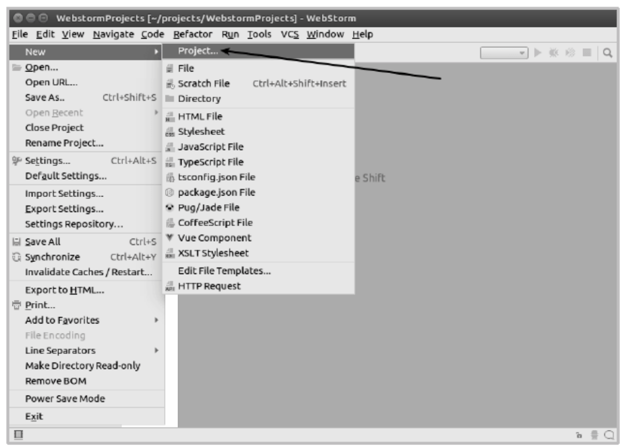


图 1.11　新建工程

如图 1.11 中箭头所指，通过菜单选项新建一个 Web 工程项目，并选择工程项目的路径，命名为 es-15days，如图 1.12 所示。

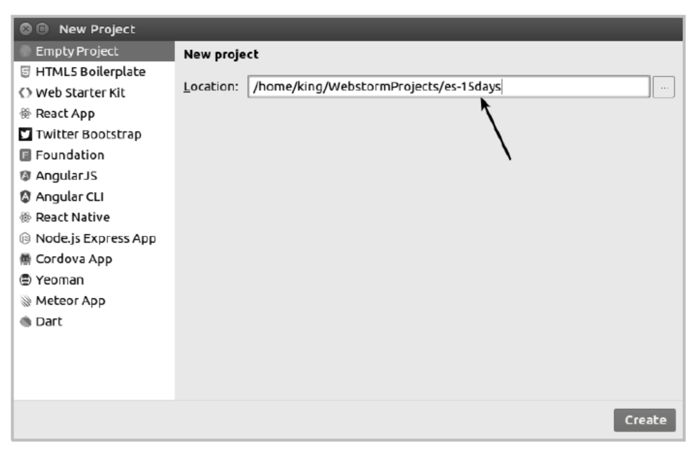


图 1.12　创建 es-15days 工程

然后，在工程项目中新建一个 HTML5 网页文件，命名为 ch01-es-debug，如图 1.13 所示。

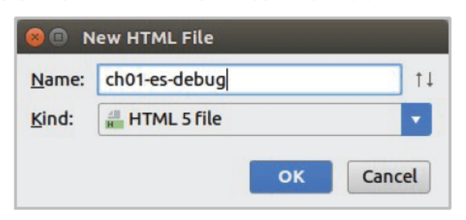


图 1.13　创建 HTML5 网页

HTML5 网页文件（ch01-es-debug.html）中定义的代码如下（详见源代码 ch01 目录中的 ch10-es-debug.html 文件）。

【代码 1-8】

```
01 <!doctype html>
02 <html lang="en">
03 <head>
04  <!-- 添加文档头部内容 -->
05  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
06  <meta http-equiv="Content-Language" content="zh-cn" />
07  <link rel="stylesheet" type="text/css" href="css/style.css">
08  <title>ECMAScript in 15-days</title>
09 </head>
10 <body>
11  <!-- 添加文档主体内容 -->
12  <header>
13      <nav>ECMAScript - 脚本代码编辑与调试</nav>
14  </header>
15  <hr>
16  <!-- 添加文档主体内容 -->
17  <div id="id-div-count">
18  </div>
19 </body>
20 <script type="text/javascript" src="js/ch01-es-debug.js"></script>
21 </html>
```

关于【代码 1-8】的分析如下：

第 17～18 行代码通过<div>标签定义了一个层元素，用于动态输出页面内容。

第 20 行代码通过<script>标签引入了外部 ECMAScript 脚本文件，其中“src”属性定义了外部脚本文件的相对路径地址（"js/ch01-es-debug.js"）。

1.4.2 第二步：使用 WebStorm 集成开发平台创建脚本文件

下面通过 WebStorm 开发平台创建【代码 1-8】中引入的“ch01-es-debug.js”脚本文件，如图 1.14 所示。

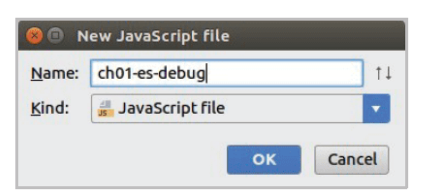


图 1.14 创建 ECMAScript 脚本文件

一般情况下不建议使用 WebStorm 集成开发平台编辑代码（因其太重量级了），建议使用 Sublime Text 这类代码编辑器编写脚本代码，如图 1.15 所示。

Sublime Text 代码编辑器简洁、快速、易于上手，编写具体代码时比使用集成开发工具有一定优势。当然，这点也是根据个人喜好因人而异的，没有高低优劣之分。

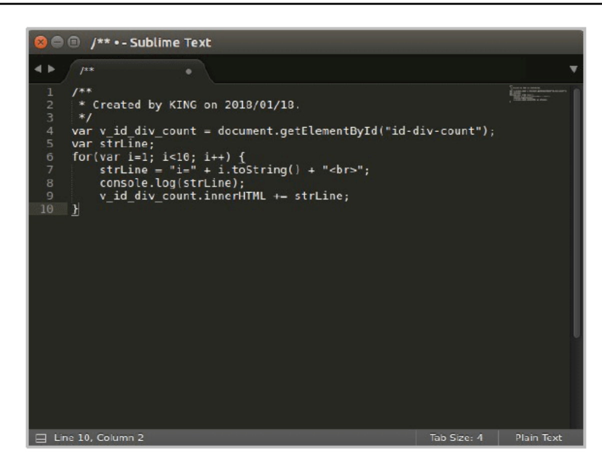


图 1.15　使用 Sublime Text 3 编写脚本代码

ECMAScript 脚本文件（ch01-es-debug.js）中定义的代码如下（详见源代码 ch01 目录中的 js/ch01-es-debug.js 文件）。

【代码 1-9】

```
01  var v_id_div_count = document.getElementById("id-div-count");
02  var strLine;
03  for(var i=1; i<=10; i++) {
04      strLine = "i=" + i.toString() + "<br>";
05      console.log(strLine);
06      v_id_div_count.innerHTML += strLine;
07  }
```

【代码 1-9】的主要功能是向 HTML 网页中循环动态写入文本，关于每行代码的具体功能含义我们在后面章节的学习中会逐步介绍。

1.4.3　第三步：使用 Firefox 浏览器运行 HTML 页面和调试脚本代码

使用 Firefox 浏览器运行【代码 1-8】定义的 HTML 网页（ch01-es-debug.html），如图 1.16 所示。

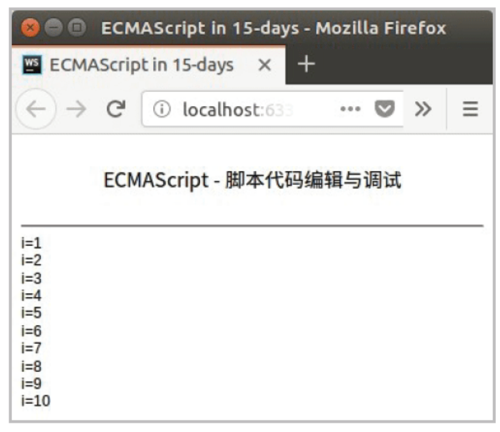


图 1.16　使用 Firefox 浏览器运行脚本代码

打开 Firefox 浏览器的调试功能面板，如图 1.17 所示。

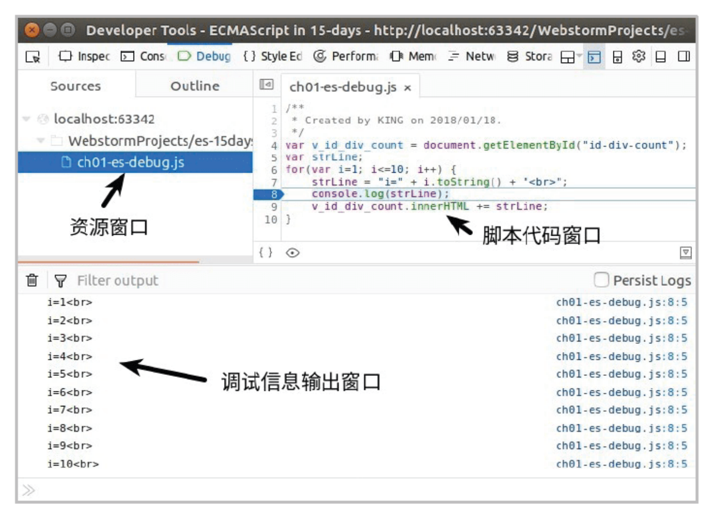


图 1.17　打开 Firefox 浏览器脚本代码调试功能面板

如图 1.17 中的箭头所指，在脚本源代码窗口中为【代码 1-9】的第 05 行脚本语句设置断点，如图 1.18 所示。

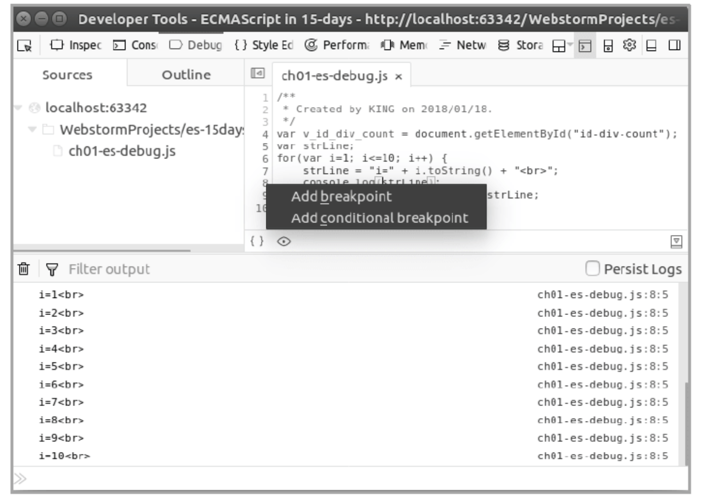


图 1.18　使用 Firefox 浏览器为脚本代码设置断点

按 F5 功能键刷新页面，再按 F11 功能键调试执行脚本代码，页面效果如图 1.19 和图 1.20 所示。

在图 1.19 和图 1.20 中可以看到，每次执行到【代码 1-9】中第 05 行脚本语句设置断点处时，脚本代码均会被中断，然后在日志窗口中输出调试信息（变量 i 计数器的数值）。以上就是 ECMAScript 脚本代码开发与调试的基本过程。

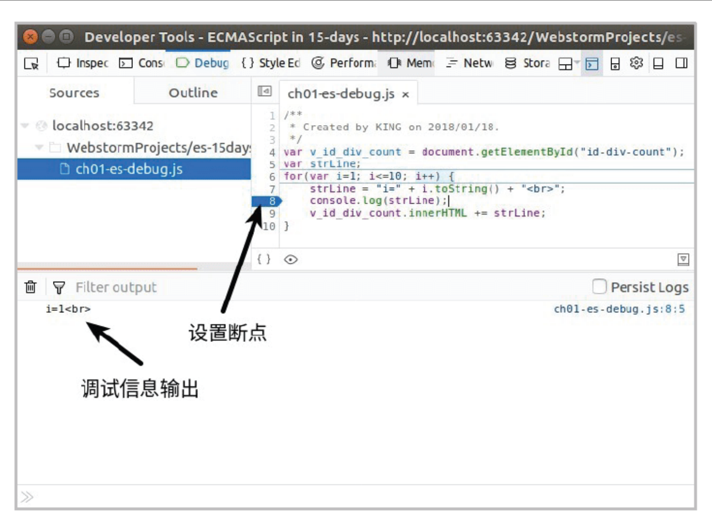


图 1.19　使用 Firefox 浏览器调试脚本代码（1）

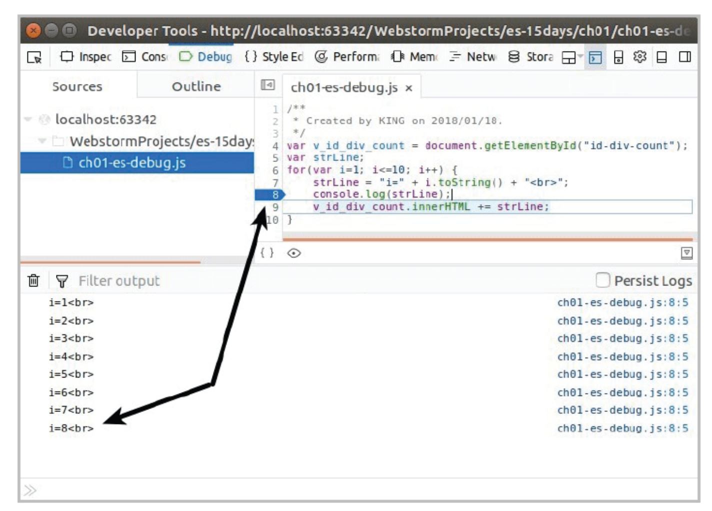


图 1.20　使用 Firefox 浏览器调试脚本代码（2）

1.5　本章小结

本章主要介绍了 ECMAScript 脚本语言的基础知识，包括 ECMAScript 的基本概念、发展历史、版本更迭、组成部分等，以及 ECMAScript 脚本代码的运行机制、编辑和调试方法等，且每个重要的知识点均通过具体的代码实例进行了讲解。希望本章内容能为读者深入学习 ECMAScript 技术做好铺垫。

第 2 章

ECMAScript 语法

学习一门编程语言，首先要掌握的就是该编程语言的语法基础。从本章开始，我们将对ECMAScript 语法进行系统全面的介绍。

2.1 ECMAScript 语法规范

学习 ECMAScript 语法主要依据的是 Ecma 国际的标准规范（ECMA-262），在该规范中详尽地定义了 ECMAScript 的所有语法。

2.1.1 ECMAScript 语句

其实，编程就如同写作一样，只不过更强调逻辑性、严谨性和技巧性。我们写一篇文章就是通过一句句的内容表述来完成表达，编写软件程序同样要通过一条条的程序语句来实现代码功能。因此，对于学习 ECMAScript 脚本语言来说，首先要掌握的一个基本概念就是 ECMAScript 语句。那么 ECMAScript 语句是如何定义的呢？

通常，一条 ECMAScript 语句用于描述一个完整的变量定义或功能操作，且每一条 ECMAScript 语句要以分号（;）结束，分号（;）是用来分割各条 ECMAScript 语句的关键字符。

对于 C、C++或 Java 这类的编程语言，语法规定也是使用分号（;）表示一条语句。不过也有另类，当前红透半边天的 Python 语言却是强制不使用任何结束符号来表示一条语句的。其实，无论语法如何规定并无高低好坏之分，仅仅是语法风格不同罢了。当然，ECMAScript 语句使用分号（;）来结束，自然就可以实现在一行中编写多条 ECMAScript 语句（不建议使用，有可能会影响代码的阅读），这一点 Python 语言肯定是无法实现的。

ECMAScript 语法规范也算是比较另类的。在 ECMA-262 标准规范中，规定了使用分号（;）

来结束一条语句并不是强制的（可选的）。如果没有使用分号（;）结束一条语句，那么 ECMAScript 语法规范会将每行代码结尾处的换行作为一条语句的结束，不过前提是没有破坏一条 ECMAScript 语句的完整功能。因此，为了避免发生不必要的歧义，建议还是老老实实地写上分号（;）作为 ECMAScript 语句的结束，这样可以有效地保证代码的可阅读性。当然，如果将来你在阅读某些代码资料时看到没有分号（;）结束的 ECMAScript 语句，千万不要大惊小怪，这也是符合 ECMA-262 标准规范的。

2.1.2　ECMAScript 大小写字母敏感

ECMAScript 语法规范中规定对代码中的大小写字母是敏感的，也就是区分大小写字母（与 Java 语法保持一致）。ECMAScript 语法规范中规定区分大小写字母的适用范围包括变量、函数名、运算符及其他一切代码。

例如，变量 id 与变量 Id 是两个不同的变量，是绝对不能等同的；同理，函数名 getElementById() 与函数名 getElementbyID()也是不同的，写成函数名 getElementbyID()则不是有效的系统函数。

2.1.3　ECMAScript 代码空格

ECMAScript 语法规范中规定自动忽略代码中多余的空格。依据这个特点，可以通过添加空格对脚本代码进行排版，达到提高代码可阅读性的目的。

2.1.4　ECMAScript 代码强制换行

ECMAScript 语法规范中规定可以在文本字符串中使用反斜杠（\）对代码行进行强制换行。例如，下面的代码就是可以被正确解析的。

```
document.write("Hello \
ECMAScript!");
```

需要注意的是，代码强制换行只限于在文本字符串中。如果将上面的代码改写成下面的形式，那么代码是无法被正确解析的。

```
document.write \
("Hello ECMAScript!"); //这条语句是错误的
```

2.1.5　ECMAScript 代码注释

ECMAScript 代码注释分为单行注释和多行注释，被注释的 ECMAScript 代码是不会被执行的。具体说明如下：

1. ECMAScript 代码单行注释

单行注释以字符串“//”开头，例如：

```
document.write("Hello ECMAScript!");   // 向浏览器中输出字符串"Hello ECMAScript!"
```

2．ECMAScript 代码多行注释

多行注释以字符串“/*”开头并以字符串“*/”结束，例如：

```
/*
* 通过 document.write()方法
* 向浏览器中输出字符串"Hello ECMAScript!"
*/
document.write("Hello ECMAScript!");
```

2.1.6 ECMAScript 代码块

代码块这个概念源自于 Java 语言，ECMAScript 语法规范中的代码块也基本上借鉴了 Java 语言。ECMAScript 代码块用于定义一组按顺序执行的语句，且全部语句被封装在大括号（{}）内。具体代码示例如下：

```
while(1) {
// TODO: ECMAScript code.
}
```

在上面这段代码中，大括号（{}）内的全部代码可以称为一个代码块。

ECMAScript 代码块是一个非常有用的概念，我们在后面介绍关于变量作用域的概念时可体会到其用途。

2.2 ECMAScript 变量

本节将介绍 ECMAScript 变量的相关知识，从现在就真正进入到 ECMAScript 语法的实际应用部分了。

2.2.1 弱类型的 ECMAScript 变量

变量 variable 是计算机高级编程语言中非常重要的概念之一。一般意义上，我们将“变量”理解为计算机程序中用于存储数据信息的容器，也可以理解为用于替代数据信息的符号。

ECMAScript 语法规范中定义的“变量”既可以存储数据信息，也可以定义为替代表达式的符号。ECMAScript 变量一般都是通过 var 关键字定义的，而且定义的都是无特定类型（也叫弱类型）的变量。因为是弱类型，所以定义好的 ECMAScript 变量可以初始化为任意类型的值，同时可以随时改变变量的数据类型。这就是弱类型变量的特点，完全不同于强类型变量（C 语言和 Java 语言的变量）。强类型变量在定义时就固定好了数据类型（如整型、浮点型、字符串型等）。

当然，一般不建议任意改变 ECMAScript 变量的数据类型，初始化定义成什么类型最好就一直沿用该类型。因为在大型 ECMAScript 代码应用中，随意改变变量类型对于管理、调试代码来说是非常崩溃的，一旦出错就很难找到出错的地方。

2.2.2　声明 ECMAScript 变量

ECMAScript 语法规范中规定一般通过 var（变量 variable 的缩写）关键字来声明定义变量，当然也可以直接声明定义变量。通常，使用 var 关键字声明定义的是局部变量，不使用 var 关键字声明定义的是全局变量。

此外，在 ECMAScript 语法规范中还规定了一些关于声明定义变量的准则，具体内容如下：

- ECMAScript 变量需要以字母开头，大小写字母均可，且对大小写字母敏感（如 a 和 A 是不同的变量）。
- ECMAScript 变量也可以用 “$” 或 “_” 符号开头。

ECMAScript 变量分为全局变量和局部变量，且二者的定义方式、作用域及用法有明显的区别。

下面是一个使用 var 关键字声明定义 ECMAScript 变量的代码示例（详见源代码 ch02 目录中的 ch02-es-var.html 文件）。

【代码 2-1】

```
01  <script type="text/javascript">
02      var i = 1;
03      var j = 2;
04      var sum = i + j;
05    var str = "Summary = ";
06    console.log(str + sum);
07  </script>
```

关于【代码 2-1】的分析如下：

第 02～03 行代码通过 var 关键字分别定义了两个变量 i 和 j，并进行了初始化赋值操作。注意，这里初始化的值均是整数类型，因为 ECMAScript 变量弱类型的特点，所以解释程序会自动为变量创建整数类型值。

第 04 行代码通过 var 关键字定义了一个变量表达式（var sum = i + j;），表达式中的变量 i 和 j 正是第 02～03 行代码中定义的，而表达式运算的结果会保存在变量 sum 中。

第 05 行代码通过 var 关键字定义了一个变量 str，并进行了初始化赋值操作。不过这里初始化的值是字符串类型，根据 ECMAScript 变量弱类型的特点，解释程序会自动为变量创建字符串类型值。

第 06 行代码通过 console.log()函数向浏览器控制台输出调试信息（字符串变量 str 的内容和表达式变量 sum 的运算结果）。

运行【代码 2-1】所指定的 HTML 页面，并使用浏览器控制台查看调试信息，页面效果如图 2.1 所示。在浏览器控制台中输出了【代码 2-1】中第 06 行脚本代码所定义的调试信息。

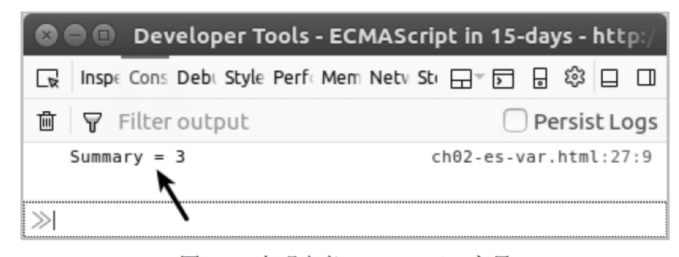


图 2.1　声明定义 ECMAScript 变量

2.2.3　ECMAScript 变量命名习惯

2.2.2 小节介绍了如何声明定义 ECMAScript 变量，本小节专门介绍 ECMAScript 变量命名的习惯。在软件编程业内有许多关于变量命名的习惯，比较著名的有 Camel 标记法（小写字母开头）、Pascal 标记法（大写字母开头）和匈牙利类型标记法（基本集合了前两者的优点）。

匈牙利类型标记法是由微软公司的匈牙利籍程序员（Charles Simonyi）首先发明出来并应用到实际开发中的。为了纪念这名具有传奇色彩的程序员，该标记法就以 Charles Simonyi 本人的名字而命名了。关于 Charles Simonyi 本人，用“传奇”来形容其在软件开发领域的功绩是一点也不夸张的，感兴趣的读者可以去互联网上搜索一下。

匈牙利类型标记法具体如何定义呢？概括来说，就是变量名由该变量所代表数据类型的小写字母缩写开始，后面由该变量代表的具体含义的单词（首字母大写，可为缩写）完成。同时，匈牙利类型标记法约定了代表数据类型的小写字母缩写，ECMAScript 语法约定的匈牙利类型标记法见表 2.1。

表 2.1　匈牙利类型标记法（ECMAScript）

类型	前缀	示例
整型（数字）	i	iValue
浮点型（数字）	f	fValue
字符串	s	sValue
数组	a	aArray
布尔型	b	bBoolean
对象	o	oObject
函数	fn	fnMethod
正则表达式	re	rePattern
变型（可以是任何类型）	v	vValue

备　注

对于匈牙利类型标记法在具体实践中是比较灵活的，读者能掌握其总体原则、领会其精要、灵活运用即可。

2.2.4　动态改变 ECMAScript 变量类型

在前面介绍了 ECMAScript 变量弱类型的特点，也就是 ECMAScript 语法规范允许在编程中动态改变 ECMAScript 变量的数据类型。简单来说，就是初始化一个 ECMAScript 变量时为一种数据类型，后面编程中还可以随时改变该变量的数据类型。这点是完全不同于强类型变量编程语言的，但也恰恰体现了 ECMAScript 脚本语言的灵活性。

下面是一个改变 ECMAScript 变量数据类型的代码示例（详见源代码 ch02 目录中的 ch02-es-var-dyn-revise.html 文件）。

【代码 2-2】

```
01  <script type="text/javascript">
```

```
02    var i = 88, str = "ECMAScript";
03    console.log("i : " + i);
04    console.log("str : " + str);
05    i = str;
06    str = 88;
07    console.log("i = " + i);
08    console.log("str = " + str);
09  </script>
```

关于【代码 2-2】的分析如下：

第 02 行代码通过 var 关键字在一行内分别定义了两个变量 i 和 str，变量 i 初始化赋值为整数类型，变量 str 初始化为字符串类型。

第 05 行代码通过表达式将变量 str 的数据内容赋给了变量 i；第 06 行代码将变量 str 的内容重新赋值为整型值 88。注意，这里的数据类型不是一致的，如果是强类型变量的编程语言（C 语言或 Java 语言），就肯定会报错，但是，ECMAScript 语法规范却是允许的，读者看到后面的调试结果就会明白。

运行页面，调试信息如图 2.2 所示。从浏览器控制台中输出的调试信息可以看到，变量 i 和变量 str 在动态改变了数据类型后并没有出现报错问题，完全可以进行正确的输出。这就是 ECMAScript 变量弱类型的特点。

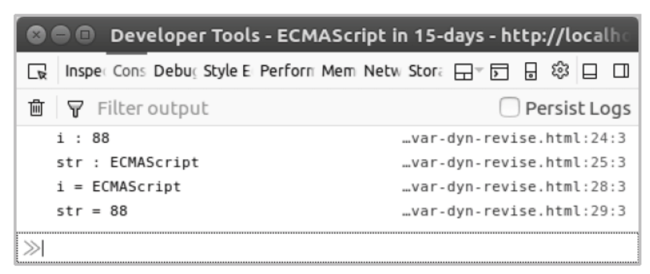


图 2.2　动态改变 ECMAScript 变量数据类型

2.2.5　未声明的 ECMAScript 变量

ECMAScript 语法规范中还规定可以不通过 var 关键字直接声明定义变量，这一点与其他程序设计语言还是有区别的。前面我们也提到了，对于未使用 var 关键字声明定义的变量一般都是全局变量。关于全局变量与局部变量的内容会在后面专门进行介绍。

下面是一个未声明的 ECMAScript 变量的代码示例（详见源代码 ch02 目录中的 ch02-es-no-var.html 文件）。

【代码 2-3】

```
01  <script type="text/javascript">
02      var sVar = "ECMAScript variable";
03      console.log("sVar : " + sVar);
04      sNoVar = sVar + " with no var";
05      console.log("sNoVar : " + sNoVar);
06  </script>
```

关于【代码 2-3】的分析如下：

第 02 和第 03 行代码通过 var 关键字定义了一个变量 sVar，并进行了初始化赋值操作。

第 04 行代码未通过 var 关键字进行声明，而是直接定义了一个变量 sNoVar，并通过变量 sVar 连接字符串的方式对其进行了初始化赋值操作。

运行页面，调试信息如图 2.3 所示。

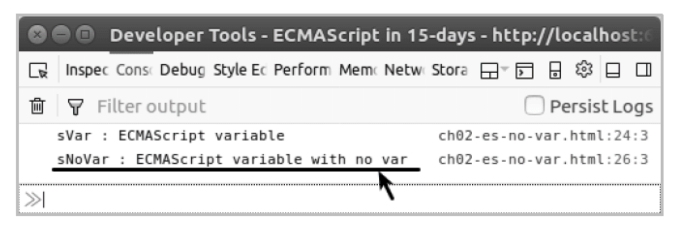


图 2.3　未声明定义的 ECMAScript 变量

如图 2.3 中箭头所指,【代码 2-3】中第 04 行脚本代码未使用 var 关键字声明定义的变量 sNoVar 同样是可以进行赋值操作的。

这里需要特别说明一下，虽然 ECMAScript 语法规范允许不使用 var 关键字声明定义变量，但这样做会带来一定的危险性。在 ECMAScript 脚本代码体量非常大的场景下，有可能会对这类未声明的变量失去监管，并产生意想不到的严重后果。因此，建议设计人员还是保持良好的编程习惯，除非在有特殊需要的情况下，否则一定要按照规范严格声明定义所有的变量。

2.2.6　严格模式

在 2.2.5 小节中详细介绍了不使用 var 关键字声明定义变量的情况，也建议设计人员最好声明定义每一个变量。因此，从 ECMAScript 5 语法规范开始，增加了一个严格模式（use strict），用来强制设计人员通过 var 关键字声明定义变量，否则调试运行时就会报错。

需要说明一下，严格模式（use strict）并不是一个运算符，仅仅是一个字面量。严格模式（use strict）并非强制使用，但强烈建议使用。使用时，“use strict”字面量一定要放置在所有脚本代码的最顶端，但如果是仅仅针对一个代码段内的代码，就要放置在该代码段的最顶端位置。

下面是一个使用严格模式（use strict）的 ECMAScript 代码示例（详见源代码 ch02 目录中的 ch02-es-var-use-strict.html 文件），这段代码是在【代码 2-3】的基础上稍作修改而成的。

【代码 2-4】

```
01  <script type="text/javascript">
02      "use strict";
03      var sVar = "ECMAScript Variable";
04      console.log("sVar : " + sVar);
05      sNoVar = sVar + " with no var";
06      console.log("sNoVar : " + sNoVar);
07  </script>
```

关于【代码 2-4】的分析如下：

代码【代码 2-4】与【代码 2-3】的唯一区别就是在第 02 行代码（ECMAScript 脚本代码的最顶端）定义了严格模式（use strict）字面量。

运行页面，调试信息如图 2.4 所示。如图 2.4 中箭头所指，【代码 2-4】第 05 行代码中未使用

var 关键字声明定义的变量 sNoVar，被浏览器报错了（undeclared variable，未声明的变量）。这个结果与【代码 2-3】运行结果是完全不同的（见图 2.3）。

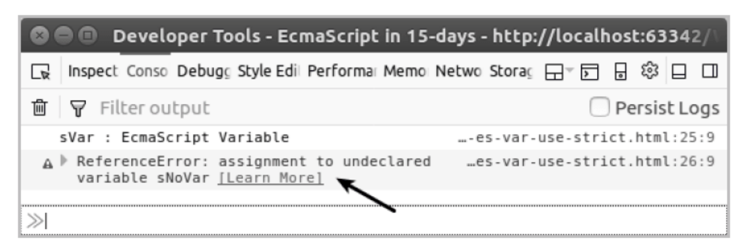


图 2.4　严格模式（use strict）

严格模式（use strict）对于编写 ECMAScript 脚本代码来说非常有用，可以有效地避免对未声明变量的使用。建议设计人员在使用变量前必须先声明定义该变量，这也是一个良好的编程习惯。

2.3　let 关键字

在 ECMAScript 6 版本语法规范中，对于变量的声明定义新增了一个 let 关键字。可能有些读者会有疑问，var 关键字不是完全可以满足声明定义变量的需求么？读者耐心学习完本节的内容，应该就能体会到 ECMAScript 6 版本语法规范中新增该功能的妙处了。

2.3.1　变量作用域

前面介绍了不通过 var 关键字声明变量的使用方法。其实，使用或不使用 var 关键字还有一个很常见的问题，那就是变量的作用域。

下面是一个 ECMAScript 变量作用域的代码示例（详见源代码 ch02 目录中的 ch02-es-var-scope.html 文件）。

【代码 2-5】

```
01  <script type="text/javascript">
02      var i = 1;                          // TODO: 声明定义变量 i
03       console.log("i = " + i);
04      /*
05       * 声明定义函数: func_i()
06       */
07      function func_i() {
08          i = 2;                      // TODO: 尝试修改变量 i 的值
09      }
10      func_i();                       // TODO: 调用函数 func_i()
11      console.log("i = " + i);
12      var j = 1;                          // TODO: 声明定义变量 j
13      console.log("j = " + j);
14      /*
15       * 声明定义函数: func_j()
16       */
```

```
17        function func_j() {
18            var j = 2;                // TODO: 重新声明定义变量 j
19        }
20        func_j();                     // TODO: 调用函数 func_j()
21        console.log("j = " + j);
22    </script>
```

关于【代码 2-5】的分析如下：

第 02 行代码通过 var 关键字定义了第一个变量 i，并进行了初始化赋值操作 i=1。第 07～09 行代码定义了一个函数方法 func_i()。其中，第 08 行代码尝试重新对变量 i 进行赋值操作 i=2。

第 10 行代码对第 07～09 行代码定义的函数方法 func_i()进行了调用。

第 12～21 行代码与第 02～11 行代码的写法类似，定义了第二个变量 j；唯一不同的地方就是第 18 行代码，使用 var 关键字重新声明定义了变量 j。

运行页面，调试信息如图 2.5 所示。

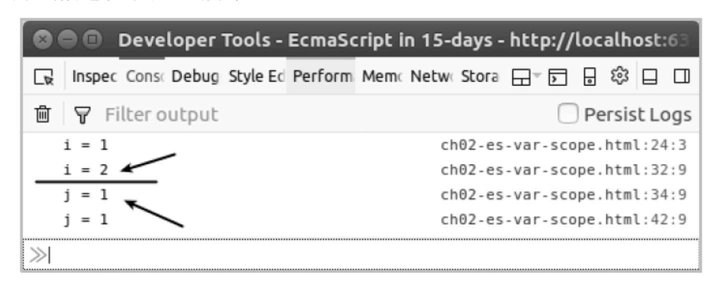


图 2.5　ECMAScript 变量作用域

如图 2.5 中箭头所指，【代码 2-5】中第 08 行代码对第一个变量 i 的修改操作成功了，变量 i 的值从 1 变成了 2。第 18 行代码对第二个变量 j 的修改操作似乎没效果，变量 j 的值依旧为 1，这是什么原因呢？

因为在第 18 行代码中使用 var 关键字重新声明定义了变量 j，此时的变量 j 仅仅是局部变量（只在函数方法 func_j()中有效），也就是说第 18 行代码定义的变量 j 与第 12 行代码定义的变量 j 是完全不相关的两个变量，虽然变量名称相同，但作用域却完全不同。

【代码 2-5】就是对变量作用域的一个简单测试，下面进一步介绍 ECMAScript 语法规范中变量提升的内容。

2.3.2　变量提升

在 ECMAScript 语法规范中，变量提升是一个很常见的情况。那么，ECMAScript 变量提升具体是一个什么概念呢？

看一下 ECMAScript 变量提升的简单测试代码（详见源代码 ch02 目录中的 ch02-es-var-enhance.html 文件）。

【代码 2-6】

```
01    <script type="text/javascript">
02        console.log("i = " + i);
03        var i = 1;
```

```
04        console.log("i = " + i);
05    </script>
```

关于【代码 2-6】的分析如下：

第 03 行代码通过 var 关键字声明定义了一个变量 i，并进行了初始化赋值操作 i=1。

第 02 行和第 04 行代码分别在声明定义变量 i 之前和之后，尝试在浏览器控制台窗口中输出了变量 i 的内容。

运行页面，调试信息如图 2.6 所示。如图 2.6 中箭头所指，虽然执行【代码 2-6】中的第 02 行代码看似会被报错，但实际却输出了变量未定义（undefined）的内容。这是为什么呢？

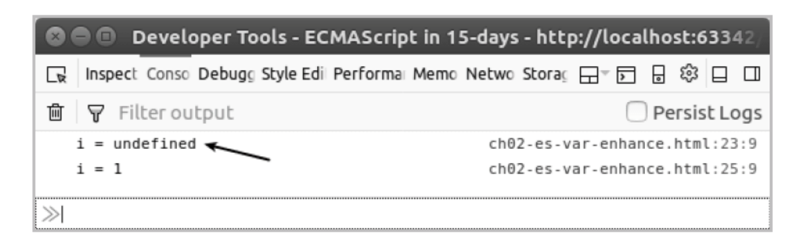


图 2.6　ECMAScript 变量提升

主要是因为 ECMAScript 变量提升的特性才会产生这样的结果。在 ECMAScript 脚本代码编译过程中，会将全部变量提升到该变量作用域的最顶部。现在我们返回到【代码 2-6】中，根据变量提升的特性，在执行第 02 行代码时变量 i 已经存在了（变量提升），只不过此时未被初始化，因此才会输出（undefined）值。在执行第 04 行代码时，变量 i 已经被初始化了，因此会输出初始化时的数值 1。

2.3.3　全局变量、局部变量和块级作用域

在 ECMAScript 6 语法规范之前，变量作用域只有全局作用域和函数作用域，是没有“块级作用域”这个概念的。因此，ECMAScript 变量只有全局变量和局部变量之分，全局变量的有效作用域就是整个页面，而局部变量的有效作用域仅限制在其所定义位置的函数内。

下面看一个 ECMAScript 全局变量、局部变量和块级作用域的代码示例（详见源代码 ch02 目录中的 ch02-es-var-block.html 文件）。

【代码 2-7】

```
01    <script type="text/javascript">
02        var global = 1;              // TODO: define global variable
03        console.log("global variable = " + global);
04        /*
05         * 定义函数作用域
06         */
07        (function () {
08            console.log("global variable in func = " + global);
09            var l = 2;
10            g = l;
11        })();
12        console.log("g = " + g);
```

```
13        /*
14         * 定义块级作用域
15         */
16        {
17            console.log("global variable in block = " + global);
18            console.log("g variable in block = " + g);
19            var ll = 3;
20            gg = ll;
21        }
22        console.log("ll = " + ll);
23        console.log("gg = " + gg);
24        console.log("l = " + l);
25    </script>
```

关于【代码 2-7】的分析如下：

这段脚本代码相对有些复杂，总体上是为了测试和验证函数作用域及块级作用域的概念。

第 02 行代码定义了一个全局变量 global，并初始化赋值为数值 1；同时，在第 03 行代码中输出了该全局变量 global 的内容。

第 07～11 行代码定义了一个函数作用域（function）。其中，第 08 行代码再次尝试输出全局变量 global 的内容。

第 09 行代码通过 var 关键字定义第一个局部变量 l，并初始化赋值为数值 2。

第 10 行代码未通过 var 关键字直接定义了第一个变量 g，并初始化赋值为变量 l 的数值；根据前面介绍过的知识点，该变量 g 是一个全局变量。

第 12 行代码尝试输出变量 g 的内容。注意，该行代码已经是在第 07～11 行代码定义的函数作用域（function）外部了。

第 16～21 行代码定义了一个块级作用域（通过“{}”符号实现）。其中，第 17 行代码第三次尝试输出全局变量 global 的内容；第 18 行代码再次尝试变量 g 的内容，注意该行代码在第 16～21 行代码定义的块级作用域内。

第 19 行代码通过 var 关键字定义了第二个局部变量 ll，并初始化赋值为数值 3。

第 20 行代码未通过 var 关键字直接定义了第二个变量 gg，并初始化赋值为变量 ll 的数值，该变量 gg 也是一个全局变量。

第 22 行代码尝试输出变量 ll 的内容，第 23 行代码尝试输出变量 gg 的内容，注意这两行代码已经是在第 16～21 行代码定义的块级作用域外部了。

比较特别的是第 24 行代码，尝试输出了变量 l 的内容。为什么将该行代码放在最后呢？通过下面的测试就会明白了。

运行页面，调试信息如图 2.7 所示。

如图 2.7 所示，从浏览器控制台中输出的内容来看，每次全局变量 global 均得到了正确输出，说明全局变量的作用域为整个页面。

由于变量 g 未通过 var 关键字声明定义，虽然是在函数作用域内定义的，也相当于一个全局变量，因此变量 g 无论是在函数作用域外部还是在块级作用域内部，均得到了正确输出。

变量 gg 的情况类似于变量 g，虽然是在块级作用域内定义的，但是也相当于一个全局变量。

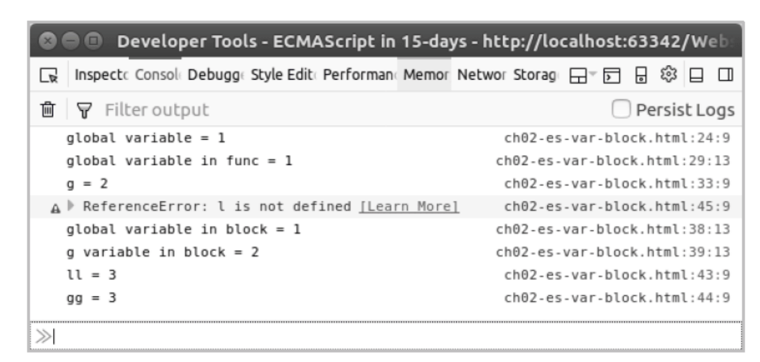


图 2.7　ECMAScript 全局变量、局部变量和块级作用域

这里比较特殊的、具有对比性的就是变量 l 和变量 ll，变量 l 通过 var 关键字在函数作用域内声明定义，变量 ll 通过 var 关键字在块级作用域内声明定义。从测试结果来看，变量 ll 得到了正确输出，而变量 l 被提示报错为“变量未定义”，由此可以看出函数作用域和块级作用域的主要区别。另外，将变量 l 的测试输出放在全部代码最后，也是为了避免因调试报错而中断脚本代码的运行。

2.3.4　let 关键字的简单示例

前面详细介绍了“块级作用域”的概念，为了更好地满足对“块级作用域”的使用，ECMAScript 6 语法规范中新增了一个 let 关键字，用来针对“块级作用域”进行编程设计。下面看一个 let 关键字的简单代码示例（详见源代码 ch02 目录中的 ch02-es-let.html 文件）。

【代码 2-8】

```
01  <script type="text/javascript">
02       "use strict";
03      /*
04       * 定义块级作用域
05       */
06      {
07          var vi = 1;
08          let li = vi;
09      }
10      console.log("vi = " + vi);
11      console.log("li = " + li);
12  </script>
```

关于【代码 2-8】的分析如下：

第 06～09 行代码定义了一个块级作用域（通过“{}”符号实现）。其中，第 07 行代码通过 var 关键字声明定义了第一个变量 vi 的内容，并初始化赋值为数值 1；第 08 行代码通过 let 关键字声明定义了第二个变量 li 的内容，并初始化赋值为变量 vi 的数值。

运行页面，调试信息如图 2.8 所示。如图 2.8 中箭头所指，变量 vi 的内容得到了正确输出，而变量 li 却被提示为错误警告“变量未定义”，这就证明了通过 let 关键字声明定义的变量 li 的有效作用域仅仅存在于第 06～09 行代码定义的“块级作用域”内。

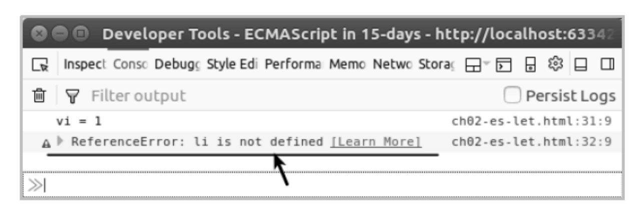


图 2.8　通过 let 关键字实现块级作用域

2.3.5　let 关键字使用规则

既然 let 关键字是 ECMAScript 6 语法规范中新增的一个特性，那么其在使用规则上自然会与 var 关键字有所区别。总体上，在使用 let 关键字时要避免出现以下两种错误情形：

- 变量在未用 let 关键字声明之前就使用会报错。
- 重复声明同一变量会报错。

下面先看第一个关于 let 关键字使用规则的代码示例（详见源代码 ch02 目录中的 ch02-es-let-rules-a.html 文件）。

【代码 2-9】

```
01  <script type="text/javascript">
02      "use strict";
03      /*
04       * 定义块级作用域
05       */
06      {
07          let i = 1;
08          console.log("i before declare= " + i);
09          console.log("j after declare = " + j);
10          let j = 2;
11      }
12  </script>
```

关于【代码 2-9】的分析如下：

第 06～11 行代码定义了一个“块级作用域”。

第 07 行代码通过 let 关键字定义了第一个变量 i，该变量的声明定义在使用之前。

第 10 行代码通过 let 关键字定义了第二个变量 j，该变量的声明定义在使用之后。

运行页面，调试信息如图 2.9 所示。从浏览器控制台中输出的结果来看，第二个变量 j 的声明定义在使用之后，因此就直接报错了。由此可见，在使用 let 关键字声明变量初始化之前是无法调用该变量的。

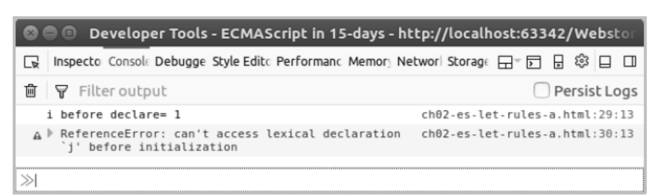


图 2.9　let 关键字使用规则（1）

下面接着看第二个关于 let 关键字使用规则的代码示例（详见源代码 ch02 目录中的 ch02-es-let-rules-b.html 文件）。

【代码 2-10】

```
01  <script type="text/javascript">
02      "use strict";
03      /*
04       * 定义块级作用域
05       */
06      {
07          let a = 1;
08          console.log("a = " + a);
09          let a = 2;
10          console.log("a = " + a);
11      }
12  </script>
```

关于【代码 2-10】的分析如下：

第 06～11 行代码定义了一个“块级作用域”。

第 07 行代码通过 let 关键字定义了一个变量 a，并进行了初始化赋值操作 a=1。

第 09 行代码再次尝试通过 let 关键字重新定义变量 a，并重新进行了初始化赋值操作 a=2。

运行页面，调试信息如图 2.10 所示。从浏览器控制台中输出的内容来看，【代码 2-10】直接报错了（提示错误为“重复声明变量”）。由此可见，使用 let 关键字是无法重复声明定义同一个变量的。

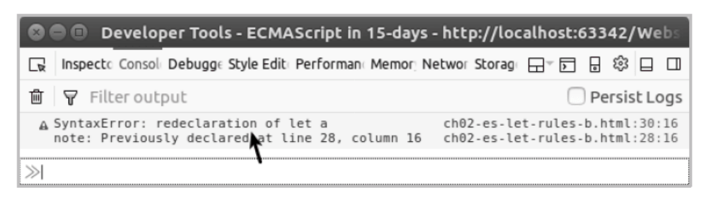


图 2.10　let 关键字使用规则（2）

2.3.6　let 关键字应用

前面介绍了这么多的铺垫内容，下面该进入重点内容了。还是之前的疑问，既然 let 关键字的功能也可以通过 var 关键字来实现，那么 ECMAScript 6 语法规范中新增的这个 let 关键字的意义何在呢？文字阐述往往没有实际代码表达得那么透彻，还是先看一个具体的代码实例。

下面是一个为了更好地介绍 let 关键字应用所进行铺垫的代码示例（详见源代码 ch02 目录中的 ch02-es-let-usage-a.html 文件）。

【代码 2-11】

```
01  <script type="text/javascript">
02        var arrJS = ["JavaScript", "ECMAScript", "TypeScript"];
03        for (var i = 0; i < 3; i++) {
04           console.log("arrJS[" + i + "] = " + arrJS[i]);
05        }
06  </script>
```

关于【代码 2-11】的分析如下：

这段代码很简单，先定义了一个字符串数组，然后通过 for 循环语句依次在浏览器控制台中输出每个数组项的内容。

运行页面，调试信息如图 2.11 所示。浏览器控制台中依次输出了每个数组项的内容。【代码 2-11】没有什么特别之处，主要是用它作为铺垫来引出下面的代码。

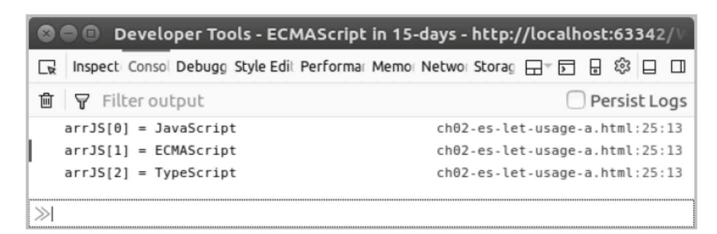


图 2.11　let 关键字应用（1）

下面接着看一个在【代码 2-11】基础上稍作改动的代码示例（详见源代码 ch02 目录中的 ch02-es-let-usage-b.html 文件）。

【代码 2-12】

```
01  <script type="text/javascript">
02        var arrJS = ["JavaScript", "ECMAScript", "TypeScript"];
03        for (var i = 0; i < 3; i++) {
04           setTimeout(function () {
05              console.log("arrJS[" + i + "] = " + arrJS[i]);
06           }, 500);
07        }
08  </script>
```

关于【代码 2-12】的分析如下：

这段代码与【代码 2-11】的主要区别就是，在第 04～06 行代码使用了 setTimeout()方法控制输出数组的内容。此处 setTimeout()方法的主要作用是设置数组的延时输出，如果读者想了解关于 setTimeout()方法的详细说明，可以参考 JavaScript 的官方文档。

运行页面，调试信息如图 2.12 所示。如图 2.12 中箭头所指，第 05 行代码输出了重复三次的数组项内容（未定义，undefined）。这个结果与图 2.11 中的内容完全不同，为什么呢？

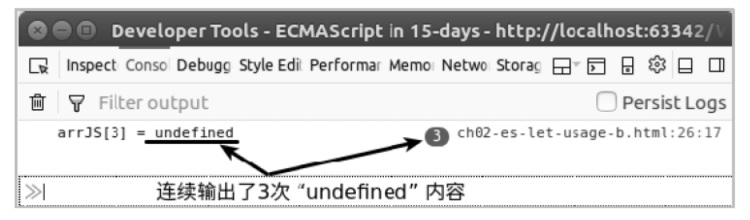


图 2.12　let 关键字应用（2）

主要原因还是变量的作用域造成的。第 03 行代码定义的 for 循环语句的自变量 i 是通过 var 关键字声明的，因此该自变量 i 的作用域就是整个脚本代码空间。由于 setTimeout()方法会设定延时，因此在 for 循环语句执行完毕后，第 05 行代码定义的在浏览器控制台中输出内容仍未执行到，而此时自变量 i 的值已经变为 3 了。所以，最后等到第 05 行代码执行时，获取的已经是数组项 arrJS[3]

（未定义，undefined）的内容了。

至于造成【代码 2-12】问题的主要原因，是由 ECMAScript 语法规范中没有“块级作用域”的概念造成的。那么该如何解决呢？此时就需要 ECMAScript 6 语法规范新增的 let 关键字发挥作用了。

下面继续看一个在【代码 2-12】基础上稍作改动的代码示例（详见源代码 ch02 目录中的 ch02-es-let-usage-c.html 文件）。

【代码 2-13】

```
01  <script type="text/javascript">
02          var arrJS = ["JavaScript", "ECMAScript", "TypeScript"];
03          for (let i = 0; i < 3; i++) {
04              setTimeout(function () {
05                  console.log("arrJS[" + i + "] = " + arrJS[i]);
06              }, 500);
07          }
08  </script>
```

关于【代码 2-13】的分析如下：

这段代码与前面【代码 2-12】的唯一区别就是第 03 行代码中的 for 循环语句的自变量 i 是通过 let 关键字来定义的。

运行页面，调试信息如图 2.13 所示。从浏览器控制台中输出的内容来看，其结果与【代码 2-11】完全相同。这就说明 let 关键字成功将自变量 i 的作用域限定在每一次 for 循环语句块内了，因此也就能获取正常的数组项的内容了。

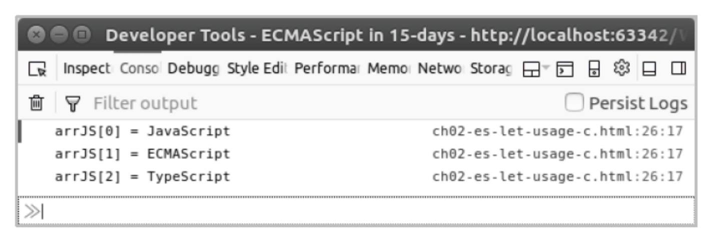


图 2.13　let 关键字应用（3）

2.4　const 关键字

常量在类似 C 和 Java 这类高级语言中很常用，ECMAScript 6 语法规范中也新增了一个 const 关键字来实现常量定义的功能。

ECMAScript 6 语法规范中的常量也适用于“块级作用域”的概念，有点像使用 let 关键字定义的变量。与其他高级语言类似，ECMAScript 的常量值同样不能通过重新赋值来改变，并且也不能重新进行声明。

常量声明的同时就要进行初始化，而运算符创建的值仅是一个只读引用，因此也无法进行更改。但也有例外，如果创建的常量是一个引用的对象，就可以改变对象的内容（如对象的参数值）了。

下面先看一下 const 的代码示例（详见源代码 ch02 目录中的 ch02-es-const.html 文件）。

【代码 2-14】

```
01  <script type="text/javascript">
02         /**
03          * const 声明时必须初始化
04          */
05         const myPI;
06  </script>
```

关于【代码 2-14】的分析如下：

第 05 行代码尝试通过 const 关键字声明了一个常量 myPI，但没有进行初始化操作。

运行页面，调试信息如图 2.14 所示。从浏览器控制台中输出的内容来看，脚本调试器直接提示需要进行常量的初始化操作。

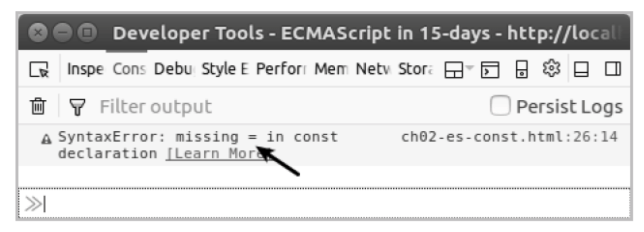


图 2.14　const 关键字（1）

下面继续看一个 const 关键字的代码示例（详见源代码 ch02 目录中的 ch02-es-const.html 文件）。

【代码 2-15】

```
01  <script type="text/javascript">
02         /**
03          * const 常量不能再进行赋值操作了
04          * @type {number}
05          */
06         const myPI = 3.1415926;
07         myPI = 3.14;
08  </script>
```

关于【代码 2-15】的分析如下：

第 06 行代码尝试对常量 myPI 再次进行赋值操作。

运行【代码 2-15】所指定的 HTML 页面，并使用浏览器控制台查看调试信息，页面效果如图 2.15 所示。如图 2.15 中箭头所指，从浏览器控制台中输出的内容来看，脚本调试器直接对常量再次赋值的初始化操作给出了重复声明定义类型的报错。

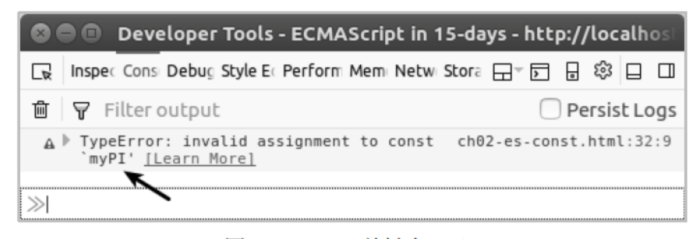


图 2.15　const 关键字（2）

接着前面的【代码 2-15】，继续看一下 const 关键字的代码示例（详见源代码 ch02 目录中的 ch02-es-const.html 文件）。

【代码 2-16】

```
01  <script type="text/javascript">
02        /**
03         * const 常量不能重复声明
04         * @type {number}
05         */
06        const myPI = 3.1415926;
07        const myPI = 3.1415926;
08  </script>
```

关于【代码 2-16】的分析如下：

第 07 行代码尝试对常量 myPI 再次进行声明操作。

运行页面，调试信息如图 2.16 所示。如图 2.16 中箭头所指，从浏览器控制台中输出的内容来看，脚本调试器还是直接对常量再次赋值的初始化操作给出了重复声明定义类型的报错。

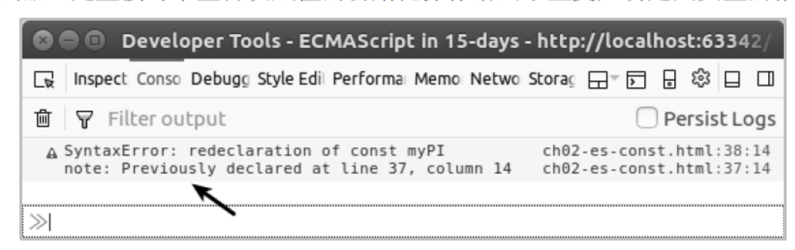


图 2.16　const 关键字（3）

另外，对于已经经过 const 关键字声明的常量，再次使用 var 或 let 关键字声明也是不被允许的。

前面的几个代码实例告诉我们，常量已经声明定义就无法再更改了，这点是毋庸置疑的。不过也有一种例外——对象常量的参数值是可以更改的。需要注意，对象常量本身是不可更改的，仅仅是常量参数可以更改。

下面看一个 const 关键字定义对象常量的代码示例（详见源代码 ch02 目录中的 ch02-es-const.html 文件）。

【代码 2-17】

```
01  <script type="text/javascript">
02        /**
03         * const 声明对象常量
04         * @type {{key: string}}
05         */
06        const myObj = {"key": "JavaScript"};
07        console.log("key : " + myObj.key);
08        myObj.key = "ECMAScript";
09        console.log("key : " + myObj.key);
10  </script>
```

关于【代码 2-17】的分析如下：

第 06 行代码通过 const 关键字声明了一个常量对象（myObj）。

第 08 行代码尝试修改常量对象 myObj 中 key 的参数值。

运行页面，调试信息如图 2.17 所示。从浏览器控制台中输出的内容来看，第 08 行代码成功修改了常量对象 myObj 中 key 的参数值。

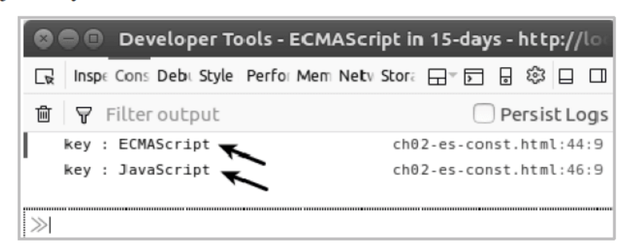


图 2.17　const 运算符（4）

2.5　ECMAScript 关键字和保留字

在 ECMAScript 语法规范中，规定了一些特殊字符串属于关键字和保留字范畴，均是不能作为变量名和函数名来使用的。下面列举一些 ECMAScript 语法规范中定义的保留字和关键字，详见表 2.2。

表 2.2　ECMAScript 关键字和保留字

ECMAScript 关键字和保留字					
abstract	arguments	boolean	break	byte	case
catch	char	class*	const*	continue	debugger
default	delete	do	double	else	enum*
export*	eval	extends*	false	final	finally
float	for	function	goto	if	implements
import*	in	instanceof	int	interface	let*
long	native	new	null	package	private
Protected	public	return	short	static	super*
switch	synchronized	this	throw	throws	transient
true	try	typeof	var	void	volatile
while	with	yield			

备注：表中带“*”的是 ECMAScript 6 中新增的。

除了表 2.2 中定义的保留字和关键字，还有一些 ECMAScript 语法规范中定义的对象、属性和方法也是避免作为变量名和函数名来使用的，具体见表 2.3。

表 2.3　ECMAScript 对象、属性和方法

ECMAScript 对象、属性和方法					
Array	Date	eval	function	hasOwnProperty	Infinity
isFinite	isNaN	isPrototypeOf	length	Math	NaN
name	Number	Object	prototype	String	toString
undefined	valueOf				

另外需要读者注意的是，如果把关键字和保留字作为变量名或函数名来使用，就有可能得到类似“Identifier Expected”这样的错误消息。

2.6　本章小结

本章主要介绍了 ECMAScript 语法规范的基础知识，包括如何通过 var 关键字声明定义变量、如何使用 let 和 const 关键字以及 ECMAScript 关键字和保留字等方面的内容，并通过一些具体实例进行讲解。希望读者通过对本章知识内容的学习，掌握 ECMAScript 脚本语言开发的基础。

第 3 章

值与类型

本章将介绍 ECMAScript 值与类型方面的相关知识，包括 ECMAScript 6 语法规范中关于类型的一些新特性，这些内容属于 ECMAScript 脚本语言中比较重要的部分。

3.1 ECMAScript 原始值与引用值

本节简单介绍ECMAScript原始值与引用值的基本内容，主要包括5种原始类型的概念和特点。

3.1.1 ECMAScript 原始值与引用值

根据 ECMA-262 规范中的定义，变量可以为两种类型的值，即原始值和引用值。那么这两种类型的值有什么区别呢？我们先看一下官方给出的原始值和引用值的定义。

- *原始值（原始类型）：原始值是存储在栈（stack）中的简单数据段，换句话解释就是原始值直接存储在变量访问的位置。根据 ECMAScript 语法规范中的介绍，原始值包括 Undefined、Null、Boolean、Number 和 String 5 大类型。*
- *引用值（引用类型）：引用值是存储在堆（heap）中的对象，简单解释就是存储在变量处的值是一个指针（pointer），指向存储对象的内存处。*

另外，这里提到了关于指针的概念，对于学习过 C 语言的读者会比较容易理解引用值的概念。如果读者对指针的概念比较模糊，建议最好选一本 C 语言教科书认真阅读一下，相信一定会有很大的助益。

3.1.2　ECMAScript 原始类型概述

ECMAScript 语法规范中定义了 5 种原始类型（primitive type），即前面提到的 Undefined、Null、Boolean、Number 和 String。根据 Ecma-262 规范中的描述，将术语“类型（type）”定义为“值的一个集合”，其中每种原始类型均定义了其所包含的值的范围及其字面量表示形式。

ECMAScript 语法规范提供了 typeof 运算符来判断一个值是否在某种类型的范围内。设计人员可以用该运算符判断一个值是否表示一种原始类型，不但可以判断出是否为原始类型，还可以判断出具体表示哪种原始类型。在 JS 脚本中使用 typeof 运算符将返回下列值之一：

- undefined：如果变量是 Undefined 类型的，会返回该类型。
- boolean：如果变量是 Boolean 类型的，会返回该类型。
- number：如果变量是 Number 类型的，会返回该类型。
- string：如果变量是 String 类型的，会返回该类型。
- object：如果变量是一种引用类型或 Null 类型的，会返回该类型。

3.2　Undefined 原始类型

首先要介绍的是 ECMAScript 语法规范中的第一种原始类型——Undefined。Undefined 类型其实只有一个值，即 undefined。当声明的变量未进行初始化时，该变量的默认值就是 undefined。

ECMAScript 语法规范中的 Undefined 类型学习起来是一个难点，通过文字描述来概括多少还是有点晦涩难懂，下面通过具体实例来理解 Undefined 原始类型的概念与用法。

先看第一个关于 Undefined 类型的代码示例（详见源代码 ch03 目录中的 ch03-es-undefined-a.html 文件）。

【代码 3-1】

```
01  <script type="text/javascript">
02       console.log("print 'undefined' is : " + undefined);
03       console.log("print 'typeof undefined' is : " + typeof undefined);
04  </script>
```

关于【代码 3-1】的分析如下：

第 02 行代码直接在浏览器控制台窗口中输出了 undefined，目的是看一下 Undefined 类型在页面中的输出效果。

第 03 行代码直接在浏览器控制台窗口中输出了“typeof undefined”，目的是看一下通过 typeof 运算符操作后的 Undefined 类型在页面中的输出效果。

运行【代码 3-1】所定义的 HTML 页面并使用浏览器控制台查看调试信息，页面效果如图 3.1 所示。

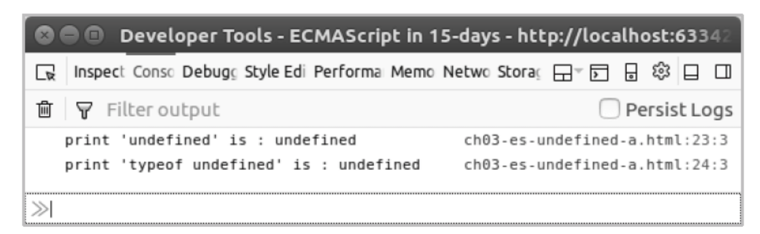


图 3.1　Undefined 原始类型（1）

从图 3.1 中可以看到，Undefined 类型和通过 typeof 运算符操作后的 Undefined 类型在控制台窗口中的输出结果均是 undefined。

下面继续看第二个关于 Undefined 类型的代码示例（详见源代码 ch03 目录中的 ch03-es-undefined-b.html 文件）。

【代码 3-2】

```
01  <script type="text/javascript">
02   var v_undefined;
03   console.log(v_undefined);
04   console.log(typeof v_undefined);
05  </script>
```

关于【代码 3-2】的分析如下：

第 02 行代码通过 var 关键字定义了一个变量 v_undefined，注意此处并未初始化该变量。

第 04 行代码直接在浏览器控制台窗口中输出了“typeof v_undefined”，目的是看一下通过 typeof 运算符操作后的变量 v_undefined 在页面中的输出效果。

运行页面，调试信息如图 3.2 所示。如果变量定义后未初始化，则无论是直接输出该变量还是通过 typeof 运算符操作后，在控制台中的输出结果均是 undefined。

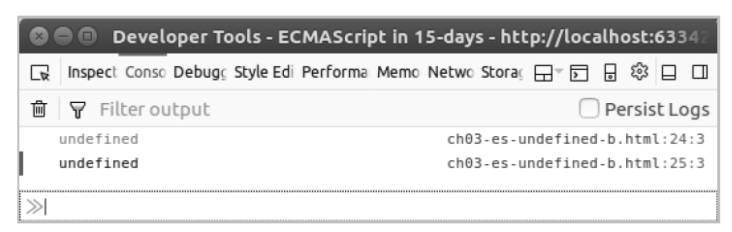


图 3.2　Undefined 原始类型（2）

下面继续看第三个关于 Undefined 类型的代码示例（详见源代码 ch03 目录中的 ch03-es-undefined-c.html 文件）。

【代码 3-3】

```
01  <script type="text/javascript">
02   console.log(typeof v_undefined);
03   console.log(v_undefined);
04  </script>
```

关于【代码 3-3】的分析如下：

第 02 行代码直接在浏览器控制台窗口中输出了“typeof v_undefined”，注意变量 v_undefined 连基本的声明定义操作都没有进行，因为我们知道 ECMAScript 语法允许变量未经过声明也可以使

用，这里的目的就是测试一下通过 typeof 运算符操作后的未声明变量 v_undefined 在页面中的输出效果。

第 03 行代码直接在浏览器控制台窗口中输出了该未声明的变量 v_undefined。

运行页面，调试信息如图 3.3 所示。如果变量未声明定义，则直接通过 typeof 运算符操作该变量后，在控制台中的输出结果仍是 undefined。如果未经过 typeof 运算符操作，直接在控制台中输出该未声明的变量（也未初始化），就会提示变量“未定义”的脚本错误。

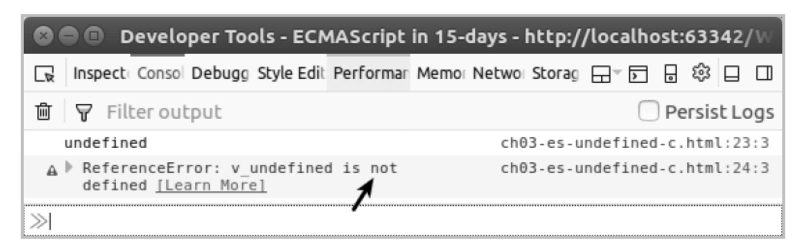


图 3.3　Undefined 原始类型（3）

下面继续看第四个关于 Undefined 类型的代码示例（详见源代码 ch03 目录中的 ch03-es-undefined-d.html 文件）。

【代码 3-4】

```
01  <script type="text/javascript">
02   var v_undefined;
03   if(v_undefined == undefined) {
04        console.log("if v_undefined == undefined is true.");
05   } else {
06        console.log("if v_undefined == undefined is false.");
07   }
08  </script>
```

关于【代码 3-4】的分析如下：

第 02 行代码通过 var 关键字定义了一个变量 v_undefined，注意此处并未初始化该变量。

第 03～07 行代码通过 if 条件运算符判断该变量 v_undefined 与 Undefined 类型是否逻辑相等，并根据逻辑运算结果在浏览器控制台窗口中输出相应的结果。

运行页面，调试信息如图 3.4 所示。从图 3.4 中箭头所指可以看到，声明后未初始化定义的变量在逻辑判断上与 Undefined 原始类型是相等的。

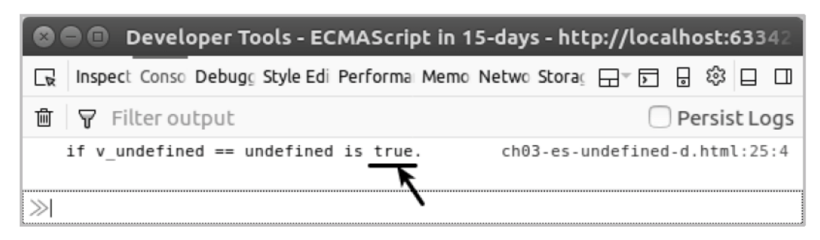


图 3.4　Undefined 原始类型（4）

最后，看一下第五个关于 Undefined 类型的代码示例（详见源代码 ch03 目录中的 ch03-es-undefined-e.html 文件）。

【代码 3-5】

```
01  <script type="text/javascript">
02        var v_func_undefined = (function(){})();
03        console.log(v_func_undefined);
04  </script>
```

关于【代码 3-5】的分析如下：

第 02 行代码通过 var 关键字定义了一个函数表达式变量 v_func_undefined，注意此处的函数并没有明确的返回值。

第 03 代码直接在浏览器控制台窗口中输出了函数表达式变量 v_func_undefined。

运行页面，调试信息如图 3.5 所示。从图 3.5 中箭头的指示可以看到，当函数未定义明确的返回值时，函数表达式变量 v_func_undefined 获取的返回值也是 undefined。

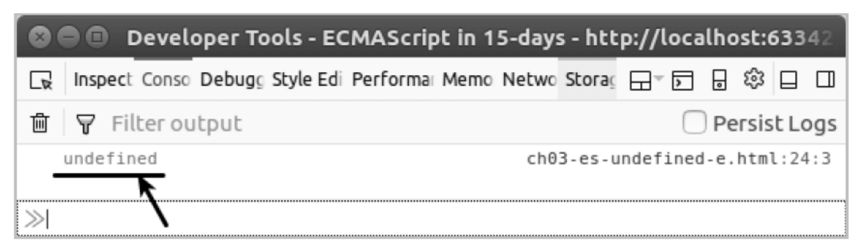


图 3.5　Undefined 原始类型（5）

3.3　Null 原始类型

本节将介绍 ECMAScript 语法规范中的第二种原始类型——Null。其实，Null 与 Undefined 类似，也是只有一个值（null）的原始类型。严格意义上来说，值 undefined 实际上是从值 null 派生而来的，正因为如此，ECMAScript 语法规范将 undefined 与 null 定义为相等的。

另外，ECMAScript 语法规范中的 Null 类型学习也是一个难点，仅仅通过文字描述来概括总结，估计难以掌握其真正的使用方法，下面还是通过具体的实例来理解 Null 原始类型的概念与用法。

下面先看第一个关于 Null 类型的代码示例（详见源代码 ch03 目录中的 ch03-es-null-a.html 文件）。

【代码 3-6】

```
01  <script type="text/javascript">
02   console.log("print null is : " + null);
03   console.log("print typeof null is : " + typeof null);
04  </script>
```

关于【代码 3-6】的分析如下：

第 02 行代码直接在浏览器控制台窗口中输出了 null，目的是看一下 Null 类型在页面中的输出效果。

第 03 行代码直接在浏览器控制台窗口中输出了“typeof null”，目的是看一下通过 typeof 运算符操作后的 Null 类型在页面中的输出效果。

运行页面，调试信息如图 3.6 所示。从图 3.6 中箭头的指示可以看到，浏览器控制台中直接输出 Null 类型的结果是 null，而通过 typeof 运算符操作后的 Null 类型，浏览器控制台中输出的结果却是 object。

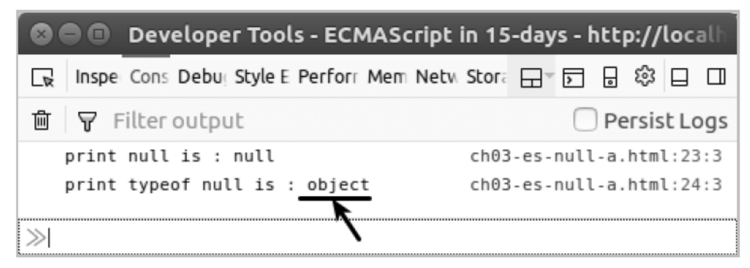


图 3.6　Null 原始类型（1）

此处读者可能就会有疑问了，为什么通过 typeof 运算符操作后的 null 值会返回 object 的结果呢？

其实，该结果源于 JavaScript 脚本语言早期版本中的一个小错误，阴差阳错又恰恰被 ECMAScript 语法规范沿用下来。目前，null 值被当作对象的占位符，这样似乎能够解释这一矛盾。但是，根据严格意义的 ECMAScript 语法规范，null 仍然是原始值，Null 也仍然是 ECMAScript 原始类型。

下面继续看第二个关于 Null 类型的代码示例（详见源代码 ch03 目录中的 ch03-es-null-b.html 文件）。

【代码 3-7】

```
01  <script type="text/javascript">
02   var v_null = null;
03   console.log(v_null);
04   console.log(typeof v_null);
05  </script>
```

关于【代码 3-7】的分析如下：

第 02 行代码通过 var 关键字定义了一个变量 v_null，同时初始化该变量的值为 null。

第 04 行代码直接在浏览器控制台窗口中输出了“typeof v_null”，目的是看一下通过 typeof 运算符操作后的变量 v_null 在页面中的输出效果。

运行页面，调试信息如图 3.7 所示。从图 3.7 中箭头的指示可以看到，浏览器控制台中直接输出了变量 v_null 类型的结果是 null，而通过 typeof 运算符操作后的变量 v_null，浏览器控制台中输出的结果是预想中的 object。

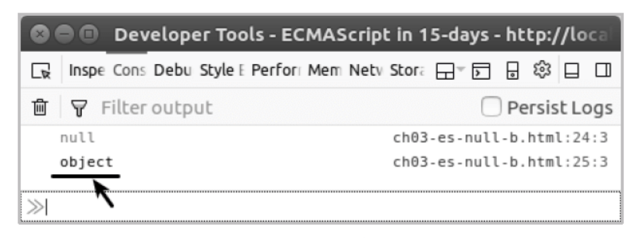


图 3.7　Null 原始类型（2）

下面再看一下 Null 类型与 Undefined 类型进行对比的代码示例（详见源代码 ch03 目录中的 ch03-es-null-c.html 文件）。

【代码 3-8】

```
01 <script type="text/javascript">
02  if(null == undefined) {
03       console.log("if null == undefined is true.");
04  } else {
05       console.log("if null == undefined is false.");
06  }
07 </script>
```

关于【代码 3-8】的分析如下：

第 02～06 行代码直接通过 if 条件运算符判断值（null）与值（undefined）是否逻辑等于（逻辑相等），并根据逻辑运算结果在浏览器控制台窗口中输出相应的信息。

运行页面，调试信息如图 3.8 所示。从图 3.8 中箭头的指示可以看到，值（null）与值（undefined）通过逻辑相等的判断结果返回值是 true。

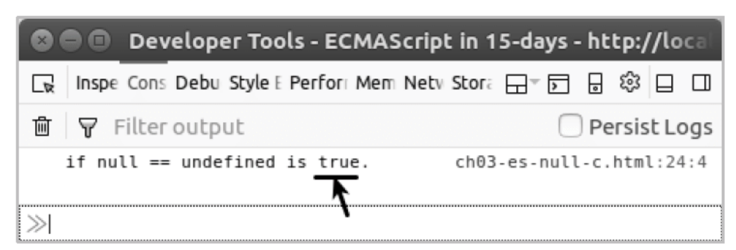


图 3.8　Null 原始类型与 Undefined 原始类型比较

尽管值（null）与值（undefined）在逻辑相等判断上是相等的，但这两个值的具体含义还是有区别的。为变量赋 null 值表示该对象目前并不存在，也可以理解为仅仅是一个空的占位符（如前文所述），而将变量定义为 undefined 值则是声明了变量但未对其进行初始化赋值（如前文代码实例）。

另外，当为函数方法定义返回的是对象类型时，若找不到该对象，则返回值通常就是 null。当尝试获取函数方法的返回值时，若该函数方法未定义返回值，则返回值通常就是 undefined。

3.4　Boolean 原始类型

本节将介绍 ECMAScript 语法规范中的第 3 种原始类型——Boolean。对于 Boolean 原始类型，是 ECMAScript 语法规范中定义的最常用的类型之一。Boolean 类型有两个值，即大家所熟知的 true 和 false。

看一下 Boolean 类型的代码示例（详见源代码 ch03 目录中的 ch03-es-boolean.html 文件）。

【代码 3-9】

```
01  <script type="text/javascript">
02       /*
```

```
03          * Boolean --- true
04          */
05         var v_b_true = true;
06         console.log(v_b_true);
07         console.log(typeof v_b_true);
08         if(v_b_true)
09            console.log("v_b_true is true.");
10         else
11            console.log("v_b_true is false.");
12         if(v_b_true == 1)
13            console.log("v_b_true == 1.");
14         else
15            console.log("v_b_true != 1.");
16        /*
17         * Boolean --- false
18         */
19         var v_b_false = false;
20         console.log(v_b_false);
21         console.log(typeof v_b_false);
22         if(v_b_false)
23            console.log("v_b_false is true.");
24         else
25            console.log("v_b_false is false.");
26         if(v_b_false == 0)
27            console.log("v_b_false == 0.");
28         else
29            console.log("v_b_false != 0.");
30    </script>
```

关于【代码 3-9】的分析如下：

第 05 行代码定义了一个变量 v_b_true，并初始化赋值为 true。

第 07 行代码直接在浏览器控制台窗口中输出了“typeof v_b_true”，目的是看一下通过 typeof 运算符操作后的 Boolean 类型在页面中的输出效果。

第 08～11 行代码直接通过 if 条件运算符判断变量 v_b_true 的逻辑值，并根据逻辑运算结果在浏览器控制台窗口中输出相应的信息。

第 12～15 行代码直接通过 if 条件运算符判断变量 v_b_true 与数值“1”是否逻辑等于，并根据逻辑运算结果在浏览器控制台窗口中输出相应的信息。这样测试的目的是源于 C 语言和 Java 语言的语法中布尔值“真”与数值“1”是逻辑相等的，此处测试一下在 ECMAScript 语法规范中是否也是如此。

第 19～29 行代码与第 05～15 行代码的功能类似，只不过测试的是 false 值，以及 false 值与数值 0 的关系。

运行页面，调试信息如图 3.9 所示。浏览器控制台中直接输出变量 v_b_true 的结果是 true，而通过 typeof 运算符操作后的变量 v_b_true 在浏览器控制台中输出的结果就是 Boolean 类型。通过 if 条件运算符判断变量 v_b_true 与数值 1 是否逻辑等于的结果表明，Boolean 值 true 是逻辑等于数值 1 的；同样，Boolean 值 false 与数值 0 是逻辑等于的。

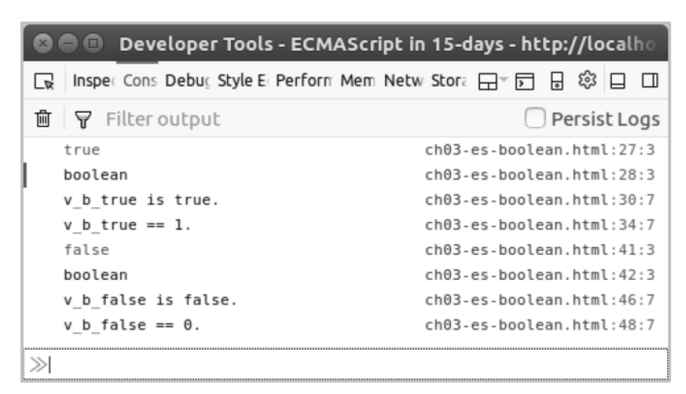


图 3.9　Boolean 原始类型

通过以上测试结果，相信读者对于 ECMAScript 语法中的 Boolean 类型的使用会有一个比较清楚的理解。

3.5　Number 原始类型

本节将介绍 ECMAScript 语法规范中的第 4 种原始类型——Number。

3.5.1　Number 原始类型介绍

Number 类型是 ECMAScript 5 种原始类型中比较重要的一种类型，同时也是比较特殊的一种类型。特殊之处就在于，Number 类型既可以表示 32 位整数值，也可以表示 64 位浮点数值。对于直接定义的任意类型数值，ECMAScript 语法规范均识别为 Number 类型的值，因此 Number 类型使用起来也是非常灵活的。

Number 类型可以表示二进制、八进制、十进制和十六进制数值，还可以表示科学计数法。同时，ECMAScript 6 语法规范中还对 Number 类型进行了扩展，增加了一些很实用的功能。

3.5.2　十进制 Number 原始类型

一般来说，高级编程语言中默认均会采用十进制数值进行运算，ECMAScript 语法规范自然也是如此设计的。

下面先看一下 Number 类型十进制数值的代码示例（详见源代码 ch03 目录中的 ch03-es-number-dec.html 文件）。

【代码 3-10】

```
01  <script type="text/javascript">
02       var i = 123;
03     console.log("i = " + i);
04     console.log("typeof i = " + typeof i);
```

```
05      console.log(i + i);
06        console.log("i + i = " + i + i);
07        var ii = "i + i = " + i + i;
08        console.log("typeof ii = " + typeof ii);
09      console.log("i + i = " + (i + i));
10       console.log("i + i = " + typeof (i + i));
11   </script>
```

关于【代码 3-10】的分析如下：

第 02 行代码声明定义了一个变量 i，并初始化赋值为十进制数值 123。

第 05 行代码尝试在浏览器控制台中输出变量 i 相加（i + i）后的结果。

第 06 行代码再次尝试在浏览器控制台中输出变量 i 相加（i + i）后的结果，不过表达方式有所不同。理论上，第 05 行和第 06 行代码似乎结果是相同的，但实际的结果呢？

运行页面，调试信息如图 3.10 所示。

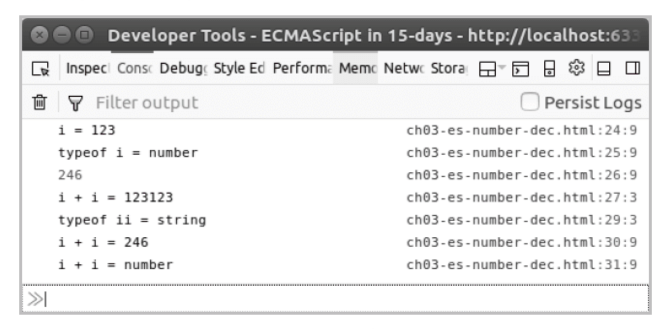


图 3.10　Number 原始类型（十进制）

从图 3.10 中可以看到，第 03～05 行代码输出的结果是符合预期的。但是第 06 行代码输出的结果（“123123”）却与第 05 行代码完全背离了，似乎是将两个变量 i 连接在了一起，为什么会这样呢？

先看第 08 行代码输出的结果，表明第 07 行代码定义的变量 ii 是一个 String 类型。原因是在 ECMAScript 语法规范中，对于将字符串与数值并列通过运算符“+”进行的操作，默认将数值自动转换为字符串，从而整体当作字符串连接操作来处理，因此第 06 行代码输出了如此结果（“123123”）。

那么，第 06 行代码将如何通过改写达到第 05 行代码的效果呢？第 09 行代码给出了答案，只需要将“i + i”用括号操作一下（“（i + i）”）即可实现，相当于提高了运算优先级。同时，第 10 行代码输出的结果也证明了“（i + i）”的 ECMAScript 类型仍是 Number 类型。

3.5.3　二进制 Number 原始类型

二进制数值是进行位运算的重要参考，ECMAScript 6 语法规范中新增了对二进制数值的定义方法。具体是通过前缀（0b 或 0B）来表示二进制数值，这里的字母“b”不区分大小写（大小写均可）。

看一下 Number 类型二进制数值的代码示例（详见源代码 ch03 目录中的 ch03-es-number-bin.html 文件）。

【代码 3-11】

```
01  <script type="text/javascript">
02      "use strict";
03      var v_bin_4 = 0b0101;  // TODO: 定义二进制数值
04      console.log("0b0101 = " + v_bin_4);
05     var v_bin_8 = 0B01010101;     // TODO: 定义二进制数值
06     console.log("0b01010101 = " + v_bin_8);
07     var v_bin_16 = 0b0101010101010101;// TODO: 定义二进制数值
08     console.log("0b0101010101010101 = " + v_bin_16);
09  </script>
```

关于【代码 3-11】的分析如下：

这段代码定义了 3 个二进制数值变量，分别实现了 4 位、8 位和 16 位二进制数值的定义过程。

运行页面，调试信息如图 3.11 所示。二进制数值输出的结果是符合预期的。另外，注意到即使变量定义为二进制数值，输出时还是要换算成十进制数值的。

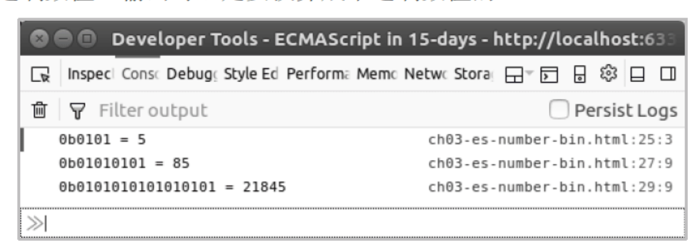


图 3.11　Number 原始类型（二进制）

3.5.4　八进制 Number 原始类型

在最早的 ECMAScript 语法规范中对八进制数值是有定义方法的，就是通过前缀数字 0 来表示。另外，在 ECMAScript 6 语法规范中还增加了对八进制数值的定义方法。具体是通过前缀（0o 或 0O）来表示八进制数值，这里的字母“o”不区分大小写（大小写均可）。

看一下 Number 类型八进制数值的代码示例（详见源代码 ch03 目录中的 ch03-es-number-oct.html 文件）。

【代码 3-12】

```
01  <script type="text/javascript">
02      var v_dec = 107;   // TODO: 定义十进制数值
03      console.log("107 = " + v_dec);
04      var v_oct = 0107;  // TODO: 定义八进制数值
05      console.log("0107 = " + v_oct);
06      var v_oct_oct = v_oct + v_oct;   // TODO: 八进制数值加法运算
07      console.log("0107 + 0107 = " + v_oct_oct);
08      var v_dec_oct = v_dec + v_oct;   // TODO: 十进制数值和八进制数值加法运算
09      console.log("107 + 0107 = " + v_dec_oct);
10     var v_oct_es6 = 0o107;   // TODO: 定义 ES6 八进制数值
11     console.log("0o107 = " + v_oct_es6);
12     if(v_oct_es6 == v_oct)
13        console.log("0o107 == 0107");
14     else
```

```
15          console.log("0o107 != 0107");
16    </script>
```

关于【代码 3-12】的分析如下：

第 02 行代码定义了第一个变量 v_dec，并初始化赋值为十进制数值 107。

第 04 行代码定义了第二个变量 v_oct，并初始化赋值为八进制数值 0107（ECMAScript 语法规范中定义八进制数值时，使用数字“0”作为开头）。注意，这里赋值使用的“0107”数值是八进制数值，与前面的十进制数值 107 是不同的。

第 06 行代码定义了第三个变量 v_oct_oct，并初始化赋值为两个八进制变量 v_oct 相加的和。

第 08 行代码定义了第四个变量 v_dec_oct，并初始化赋值为十进制变量 v_dec 与八进制变量 v_oct 相加的和。

第 10 行代码定义了第五个变量 v_oct_es6，并初始化赋值为八进制数值 0o107（ECMAScript 6 语法规范中对八进制数值重新进行了定义，使用数字字母“0o”组合作为开头，分别是数字 0 和小写字母 o）。注意，这个八进制数值的新定义只有在“严格模式”下才会遵循 ECMAScript 6 语法规范。

第 12～15 行代码通过 if 条件判断语句对八进制数值 0107 和 0o107 进行了逻辑相等的判断。

运行页面，调试信息如图 3.12 所示。

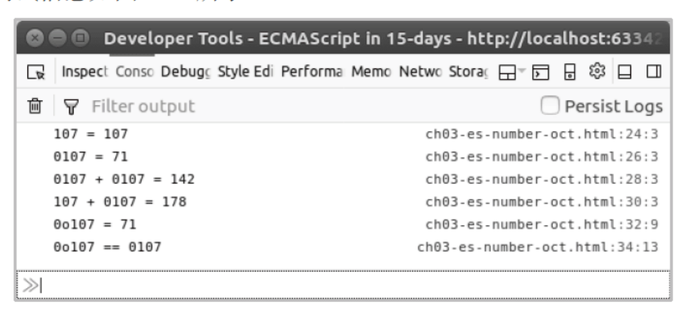


图 3.12　Number 原始类型（八进制）

从图 3.12 所示，第 05 行代码在浏览器控制台窗口中输出的变量 v_oct 的内容为 71，而不是初始化定义的数值 0107，说明 ECMAScript 语法规范对八进制变量返回的是换算后的十进制数值。因此，第 07 行代码在浏览器控制台窗口中输出的变量 v_oct_oct 的内容为 142，也是十进制相加运算得出的结果。

另外，第 09 行代码在浏览器控制台窗口中输出的变量 v_dec_oct 的内容为 178，是十进制数值 107 和十进制数值 71 相加运算得出的结果。这就表明在 ECMAScript 语法规范中八进制与十进制运算时，默认都是换算成十进制来计算的。八进制数值运算规则如此，下面的十六进制数值运算也是一样的。

最后，通过第 12～15 行代码输出的结果来看，八进制数值 0107 和 0o107 是逻辑相等的。注意，这是在非“严格模式”条件下才会成立的。如果在“严格模式”下，八进制数值 0107 是会报出错误的。

3.5.5　十六进制 Number 原始类型

通常计算机的编码系统会采用十六进制数值来表示，如 ASCII 字符编码、汉字编码、UTF 统

一编码等。ECMAScript 语法规范中对十六进制数值也是有定义方法的，就是通过前缀“0x”来表示。

看一下 Number 类型十六进制数值的代码示例（详见源代码 ch03 目录中的 ch03-es-number-hex.html 文件）。

【代码 3-13】

```
01  <script type="text/javascript">
02      var v_dec = 1234;
03      console.log("1234 = " + v_dec);
04      var v_oct = 0147;       // TODO: 定义八进制数值
05      console.log("0147 = " + v_oct);
06      var v_hex = 0x12ff;     // TODO: 定义十六进制数值
07      console.log("0x12ff = " + v_hex);
08      var v_hex_hex = v_hex + v_hex;   // TODO: 十六进制数值加法运算
09      console.log("0x12ff + 0x12ff = " + v_hex_hex);
10      var v_dec_hex = v_dec + v_hex;   // TODO: 十进制数值和十六进制数值加法运算
11      console.log("1234 + 0x12ff = " + v_dec_hex);
12      var v_oct_hex = v_oct + v_hex;   // TODO: 八进制数值和十六进制数值加法运算
13      console.log("0147 + 0x12ff = " + v_oct_hex);
14  </script>
```

关于【代码 3-13】的分析如下：

第 02 行代码定义了第一个变量 v_dec，并初始化赋值为十进制数值 1234。

第 04 行代码定义了第二个变量 v_oct，并初始化赋值为八进制数值 0147。

第 06 行代码定义了第三个变量 v_hex，并初始化赋值为十六进制数值 0x12ff（ECMAScript 语法规范中定义十六进制数值时，使用数字“0x”开头来表示）。

第 08 行代码定义了第四个变量 v_hex_hex，并初始化赋值为两个十六进制变量 v_hex 相加的和。

第 10 行代码定义了第五个变量 v_dec_hex，并初始化赋值为十进制变量 v_dec 与十六进制变量 v_hex 相加的和。

第 12 行代码定义了第六个变量 v_oct_hex，并初始化赋值为八进制变量 v_oct 与十六进制变量 v_hex 相加的和。

运行页面，调试信息如图 3.13 所示。

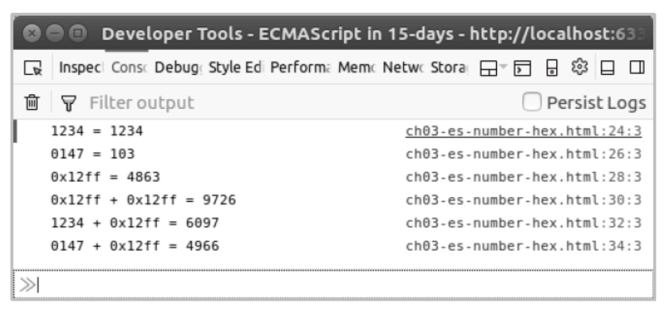


图 3.13　Number 原始类型（十六进制）

如图 3.13 所示，第 07 行代码在浏览器控制台窗口中输出的变量 v_hex 的内容为 4863，而不是初始化定义的数值 0x12ff，说明 ECMAScript 语法规范中对十六进制变量返回的同样是换算后的十进制数值。因此，第 09 行代码在浏览器控制台窗口中输出的变量 v_hex_hex 的内容为 9726，也是十进制相加运算得出的结果。

另外，第 11 行代码在浏览器控制台窗口中输出的变量 v_dec_hex 的内容为 6097，是十进制数值 1234 和十进制数值 4863 相加运算得出的结果，这就表明在 ECMAScript 语法规范中，十进制与十六进制运算时，默认都是换算成十进制来计算的。同样，第 13 行代码在浏览器控制台窗口中输出的变量 v_oct_hex 的内容为 4966，是十进制数值 103 和十进制数值 4863 相加运算得出的结果，同样表明在 ECMAScript 语法规范中，八进制与十六进制运算时，默认都是换算成十进制来计算的。

3.5.6　浮点数 Number 原始类型

在 ECMAScript 语法规范中对浮点数也有详细的定义。浮点数在运算过程中要特别注意与字符串的区别。

看一下 Number 类型浮点数值的代码示例（详见源代码 ch03 目录中的 ch03-es-number-float.html 文件）。

【代码 3-14】

```
01  <script type="text/javascript">
02      // TODO: 定义浮点数值必须使用小数点和至少一位小数
03      var v_f = 16.8;                // TODO: 定义浮点数
04      console.log("16.8 = " + v_f);
05      var v_f_f = v_f + v_f;         // TODO: 浮点数加法运算
06      console.log("16.8 + 16.8 = " + v_f_f);
07      var v_i = 168;                 // TODO: 定义整数
08      var v_i_f = v_i + v_f;         // TODO: 整数和浮点数加法运算
09      console.log("168 + 16.8 = " + v_i_f);
10      var v_str = "16.8";            // TODO: 定义浮点数字符串
11      var v_f_str = v_f + v_str;  // TODO: 浮点数和浮点数字符串加法运算
12      console.log("16.8 + '16.8' = " + v_f_str);
13  </script>
```

关于【代码 3-14】的分析如下：

第 03 行代码定义了第一个变量 v_f，并初始化赋值为浮点数值 16.8。这里需要注意一点，ECMAScript 语法规范中规定在定义浮点数值时，必须使用小数点和小数点后至少一位小数。

第 05 行代码定义了第二个变量 v_f_f，并初始化赋值为两个浮点数变量 v_f 相加的和。

第 07 行代码定义了第三个变量 v_i，并初始化赋值为整数值 168。

第 08 行代码定义了第四个变量 v_i_f，并初始化赋值为整数值变量 v_i 和浮点数变量 v_f 相加的和。

第 10 行代码定义了第五个变量 v_str，并初始化赋值为字符串“16.8”。

第 11 行代码定义了第六个变量 v_f_str，并使用运算符“+”将其初始化赋值为浮点数变量 v_f 与字符串变量 v_str 连接后的结果。

运行页面，调试信息如图 3.14 所示。

从图 3.14 中可以看到，第 06 行代码在浏览器控制台窗口中输出的变量 v_f_f 的内容为 33.6，正是两个浮点数 16.8 相加后的结果。第 09 行代码在浏览器控制台窗口中输出的变量 v_i_f 的内容为 184.8，正是整数 168 和浮点数 16.8 相加后的结果，且整数与浮点数运算后自动保存为浮点数类型。

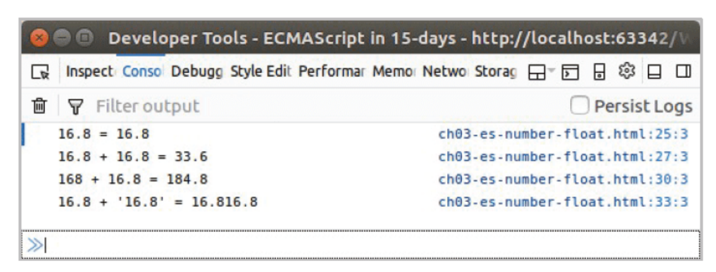


图 3.14　Number 原始类型（浮点数）

另外，第 12 行代码在浏览器控制台窗口中输出的内容为“16.816.8”，这明显是一个字符串类型。该结果表明在 ECMAScript 语法规范中，浮点数与字符串通过运算符“+”连接时，浮点数会被当成字符串来处理，而此时的运算符“+”不再是加法运算，而是字符串连接运算。

3.5.7　Number 原始类型科学计数法

本小节将介绍关于 Number 类型中非常大或非常小的数值。一般采用科学计数法来表示浮点数，具体说就是把一个很大或很小的数值表示为数字（包括十进制数字）加 e（或 E），后面加乘以 10 的幂。

下面就是关于 Number 类型科学计数法的代码示例（详见源代码 ch03 目录中的 ch03-es-number-e.html 文件）。

【代码 3-15】

```
01  <script type="text/javascript">
02   // TODO: 科学计数法使用 e 加上 10 的倍数来表示
03   var v_e_plus = 1.68e8;            // TODO: 很大的数的表示方法
04   console.log("1.68e8 = " + v_e_plus);
05   var v_e_neg_6 = 0.000000168;          // TODO: 很小的数的表示方法
06   console.log("0.000000168 = " + v_e_neg_6);
07   var v_e_neg_5 = 0.00000168;       // TODO: 很小的数的表示方法
08   console.log("0.000000168 = " + v_e_neg_5);
09  </script>
```

关于【代码 3-15】的分析如下：

第 03 行代码定义了第一个变量 v_e_plus，并初始化赋值为科学计数法的浮点数值 1.68e8。

第 05 行代码定义了第二个变量 v_e_neg_6，并初始化赋值为浮点数数值 0.000000168，这是一个很小的浮点数。

第 07 行代码定义了第三个变量 v_e_neg_5，并初始化赋值为浮点数数值 0.00000168，同样是一个很小的浮点数。

注意，变量 v_e_neg_6 与变量 v_e_neg_5 的区别就是小数点后的前导 0 的个数不同，变量 v_e_neg_6 是 6 个 0，变量 v_e_neg_5 是 5 个 0。

运行页面，调试信息如图 3.15 所示。第 04 行代码在浏览器控制台窗口中输出的变量 v_e_plus 的内容为 168000000，正是科学计数法 1.68e8（1.68×10^8）通过运算后的结果。第 06 行代码在浏览器控制台窗口中输出的变量 v_e_neg_6 的内容为 1.68e-7，正是浮点数 0.000000168 转换为科学计数法后的结果。第 08 行代码在浏览器控制台窗口中输出的变量 v_e_neg_5 的内容并没有转换为科

学计数法，这是因为 ECMAScript 语法规定，默认会把具有 6 个或 6 个以上前导 0 的浮点数自动转换成科学计数法。

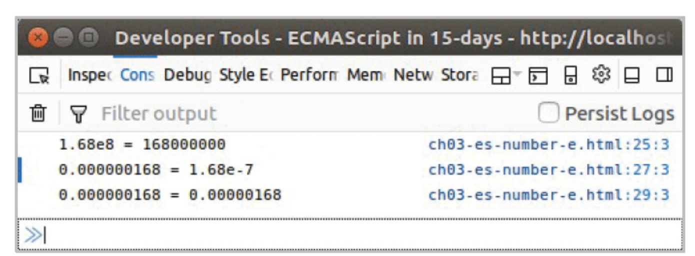


图 3.15　Number 原始类型（科学计数法）

3.6　Number 特殊值及方法

本节将介绍 ECMAScript 语法规范中为 Number 类型所定义的特殊值及方法，有些特殊值及方法是 ECMAScript 6 语法规范中新增的。

3.6.1　Number 最大值与最小值

ECMAScript 语法规范中为 Number 类型分别定义了一个最大值和一个最小值，具体就是 Number.MAX_VALUE 和 Number.MIN_VALUE 这两个特殊值，其分别定义了 Number 值集合的上下界限。

根据 ECMAScript 语法规范中的描述，所有定义的数值都必须介于这两个值之间。不过，如果是通过计算生成的数值，就可以不在这两个值之间。

看一下特殊值 Number.MAX_VALUE 和 Number.MIN_VALUE 的代码示例（详见源代码 ch03 目录中的 ch03-es-number-min-max.html 文件）。

【代码 3-16】

```
01  <script type="text/javascript">
02      // TODO: Number 类型上限值
03       console.log("Number 类型上界限 : " + Number.MAX_VALUE);
04      // TODO: Number 类型下限值
05       console.log("Number 类型下界限 : " + Number.MIN_VALUE);
06  </script>
```

关于【代码 3-16】的分析如下：

这段代码直接在浏览器控制台窗口中输出了 Number.MAX_VALUE 和 Number.MIN_VALUE 这两个特殊值的内容。

运行页面，调试信息如图 3.16 所示。浏览器控制台中输出了特殊值 Number.MAX_VALUE 和 Number.MIN_VALUE 的具体值。

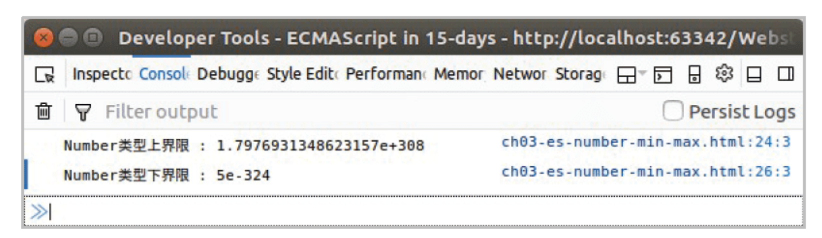


图 3.16　Number 类型特殊值（上下界限）

3.6.2　Number 无穷大

根据 ECMAScript 语法规范中的描述，当通过计算生成的数值大于 Number.MAX_VALUE 时，其会被赋予特殊值 Number.POSITIVE_INFINITY，该值表示正无限大的数值，也就是不再有具体的数值。同样地，当通过计算生成的数值小于 Number.MIN_VALUE 时，其会被赋予特殊值 Number.NEGATIVE_INFINITY，该值表示负无限大的数值，同样也是不再有具体的数值。当通过计算返回的是无穷大值时，该值也就不能再用于其他计算了。

另外，ECMAScript 语法规范中有专用值表示正无穷大（Infinity）和负无穷大（-Infinity）。事实上，Number.POSITIVE_INFINITY 的值就是 Infinity，Number.NEGATIVE_INFINITY 的值就是 -Infinity。

看一下特殊值 Number.POSITIVE_INFINITY 和 Number.NEGATIVE_INFINITY 的代码示例（详见源代码 ch03 目录中的 ch03-es-number-posi-nega-infinity.html 文件）。

【代码 3-17】

```
01  <script type="text/javascript">
02      // TODO: Number 类型正无穷大
03      console.log("Number.POSITIVE_INFINITY(正无穷大) : ");
04     console.log(Number.POSITIVE_INFINITY);
05     // TODO: Number 类型负无穷大
06      console.log("Number.NEGATIVE_INFINITY(负无穷大) : ");
07     console.log(Number.NEGATIVE_INFINITY);
08  </script>
```

关于【代码 3-17】的分析如下：

这段代码直接在浏览器控制台窗口中输出了 Number.POSITIVE_INFINITY 和 Number.NEGATIVE_INFINITY 这两个特殊值的内容。

运行页面，调试信息如图 3.17 所示。浏览器控制台中输出了特殊值 Number.POSITIVE_INFINITY 和 Number.NEGATIVE_INFINITY 的具体值，分别是特殊值正无穷大（Infinity）和负无穷大（-Infinity）。

无穷大数既可以是正数也可以是负数，那有没有办法快速判断出一个数值是不是无穷大呢？答案是肯定的。在 ECMAScript 语法规范中提供了一个 isFinite()函数方法，可以判断一个数值是否为无穷大。

看一下通过 isFinite()函数方法判断数值是否为无穷大的代码示例（详见源代码 ch03 目录中的 ch03-es-number-isInfinity.html 文件）。

Developer Tools - ECMAScript in 15-days - http://localhost:63342/
Inspect Conso Debugg Style Edit Performa Memo Netwo Storag
Filter output　Persist Logs
Number.POSITIVE_INFINITY(正无穷大) :　…posi-nega-infinity.html:24:3
Infinity　…posi-nega-infinity.html:25:9
Number.NEGATIVE_INFINITY(负无穷大) :　…posi-nega-infinity.html:27:3
-Infinity　…posi-nega-infinity.html:28:9

图 3.17　Number 类型特殊值（无穷大）

【代码 3-18】

```
01  <script type="text/javascript">
02   // TODO: Number 类型 --- isFinite()方法判断无穷大
03   if(isFinite(1)) {
04         console.log("isFinite(1) is not Infinity.");
05   } else {
06         console.log("isFinite(1) is Infinity.");
07   }
08   if(isFinite(Number.MAX_VALUE)) {
09         console.log("isFinite(Number.MAX_VALUE) is not Infinity.");
10   } else {
11         console.log("isFinite(Number.MAX_VALUE) is Infinity.");
12   }
13   if(isFinite(Number.MAX_VALUE * 2)) {
14         console.log("Number.MAX_VALUE * 2 = " + Number.MAX_VALUE * 2);
15         console.log("isFinite(Number.MAX_VALUE * 2) is not Infinity.");
16   } else {
17         console.log("Number.MAX_VALUE * 2 = " + Number.MAX_VALUE * 2);
18         console.log("isFinite(Number.MAX_VALUE * 2) is Infinity.");
19   }
20   if(isFinite(Number.POSITIVE_INFINITY)) {
21         console.log("isFinite(Number.POSITIVE_INFINITY) is not Infinity.");
22   } else {
23         console.log("isFinite(Number.POSITIVE_INFINITY) is Infinity.");
24   }
25  </script>
```

关于【代码 3-18】的分析如下：

这段代码通过 isFinite()函数方法分别判断数值 1、特殊值 Number.MAX_VALUE、两倍的特殊值 Number.MAX_VALUE 和特殊值 Number.POSITIVE_INFINITY 是否为非无穷大。

运行页面，调试信息如图 3.18 所示。数值 1 为非无穷大，特殊值 Number.MAX_VALUE 同样也为非无穷大，而两倍的特殊值 Number.MAX_VALUE 和特殊值 Number.POSITIVE_INFINITY 则为无穷大。

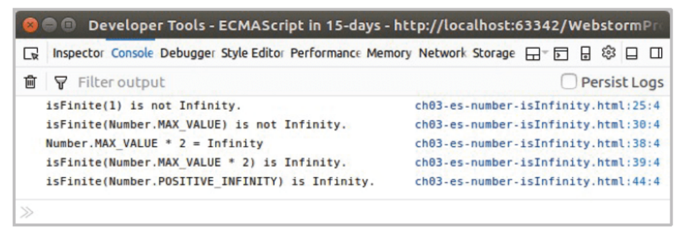


图 3.18　Number 类型特殊值（判断是否为非无穷大）

3.6.3 非数值 NaN

这里还要特别介绍一个 Number 类型特殊值——NaN，表示非数值（Not a Number）。在 ECMAScript 语法规范中，NaN 是一个非常奇怪的特殊值。特殊值 NaN 的奇怪之处就是其与自身逻辑判断上是不相等的。同样，NaN 与 Infinity 都是不能用于算术计算的。另外，ECMAScript 语法规范中提供了一个 isNaN()函数方法，可以用于判断某个数据类型是否为非数值。

下面看一个特殊值 NaN 和 isNaN()函数方法的代码示例（详见源代码 ch03 目录中的 ch03-es-isNaN.html 文件）。

【代码 3-19】

```
01 <script type="text/javascript">
02  console.log("NaN is " + NaN);
03  console.log("typeof NaN is " + typeof NaN);
04  if(isNaN(NaN)) {
05       console.log("isNaN(NaN) return true.");
06  } else {
07       console.log("isNaN(NaN) return false.");
08  }
09  if(isNaN(123)) {
10       console.log("isNaN(123) return true.");
11  } else {
12       console.log("isNaN(123) return false.");
13  }
14  if(isNaN("123")) {
15       console.log("isNaN('123') return true.");
16  } else {
17       console.log("isNaN('123') return false.");
18  }
19  if(isNaN("abc")) {
20       console.log("isNaN('abc') return true.");
21  } else {
22       console.log("isNaN('abc') return false.");
23  }
24  if(NaN == NaN) {
25       console.log("NaN == NaN return true.");
26  } else {
27       console.log("NaN == NaN return false.");
28  }
29 </script>
```

关于【代码 3-19】的分析如下：

第 02 行代码直接在浏览器控制台窗口中输出了特殊值 NaN 的内容。

第 03 行代码通过 typeof 运算符对特殊值 NaN 进行了操作，并在浏览器控制台窗口中输出了运算后的结果。

第 04～08 行代码通过 isNaN()函数方法判断特殊值 NaN 是否为非数值，并根据判断结果在浏览器控制台窗口中输出相应的信息。

第 09～13 行代码通过 isNaN()函数方法判断数值 123 是否为非数值，并根据判断结果在浏览器控制台窗口中输出相应的信息。

第 14～18 行代码通过 isNaN()函数方法判断字符串“123”是否为非数值，并根据判断结果在浏览器控制台窗口中输出相应的信息。

第 19～23 行代码通过 isNaN()函数方法判断字符串“abc”是否为非数值，并根据判断结果在浏览器控制台窗口中输出相应的信息。

第 24～28 行代码通过 if 语句判断特殊值 NaN 自身是否为逻辑相等，并根据判断结果在浏览器控制台窗口中输出相应的信息。

运行页面，调试信息如图 3.19 所示。

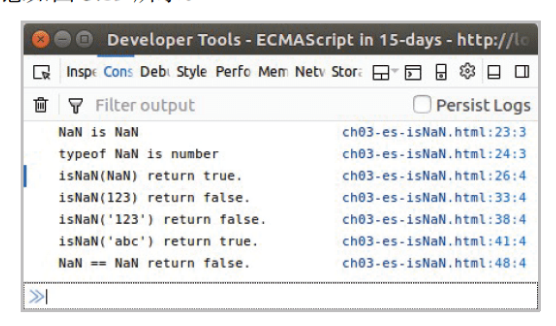


图 3.19　Number 类型特殊值（NaN）

从图 3.19 中可以看到，第 03 行代码通过 typeof 运算符操作特殊值 NaN 后结果为 Number 类型，这与 NaN 的定义是一致的。

第 04～08 行代码通过 isNaN()函数方法判断特殊值 NaN 是否为非数值的结果为 true，表示 NaN 为非数值。

第 09～13 行代码通过 isNaN()函数方法判断数值 123 是否为非数值的结果为 false，表示 123 为不是非数值。

第 14～18 行代码通过 isNaN()函数方法判断字符串“123”是否为非数值的结果为 false，表示“123”同样为不是非数值。

第 19～23 行代码通过 isNaN()函数方法判断字符串“abc”是否为非数值的结果为 true，表示“abc”为非数值。

第 24～28 行代码通过 if 语句判断特殊值 NaN 自身是否为逻辑相等的结果为 false，表示特殊值 NaN 与其自身逻辑不相等，这就是前面提到的特殊值 NaN 奇怪之处的测试结果。

3.6.4　Number 安全整数值

本小节将介绍 ECMAScript 6 语法规范中新增的一组特殊值——安全整数值。安全整数值通过 Number.MAX_SAFE_INTEGER 和 Number.MIN_SAFE_INTEGER 这两个特殊值来定义。

在早期版本的 ECMAScript 语法规范中，整数值范围为-2^53~2^53（注意是闭区间），如果超出这个范围就无法精确表示。ECMAScript 6 语法规范中新增的安全整数值就是为了表示这两个界限值。

另外，ECMAScript 6 语法规范为了配合安全整数值的使用还新增了一个 Number.isSafeInteger()函数方法，用于判断整数值是否在安全整数值范围内。

看一下安全整数值的代码示例（详见源代码 ch03 目录中的 ch03-es-safe-integer.html 文件）。

【代码 3-20】

```
01  <script type="text/javascript">
02        "use strict";
03      // TODO: Number 类型上限值
04        console.log("Number 类型安全值上界限 : " + Number.MAX_SAFE_INTEGER);
05        if(Number.MAX_SAFE_INTEGER == Math.pow(2, 53) - 1)
06            console.log("Number.MAX_SAFE_INTEGER == Math.pow(2, 53) - 1");
07        else
08          console.log("Number.MAX_SAFE_INTEGER != Math.pow(2, 53) - 1");
09      // TODO: Number 类型下限值
10        console.log("Number 类型安全值下界限 : " + Number.MIN_SAFE_INTEGER);
11      if(Number.MIN_SAFE_INTEGER == 1 - Math.pow(2, 53))
12          console.log("Number.MIN_SAFE_INTEGER == 1 - Math.pow(2, 53)");
13      else
14          console.log("Number.MIN_SAFE_INTEGER != 1 - Math.pow(2, 53)");
15      // TODO: Number.isSafeInteger() 方法用于判断是否在安全值范围内
16        if(Number.isSafeInteger(Math.pow(2, 53)))
17          console.log("Math.pow(2, 53) is a safe Integer.");
18        else
19          console.log("Math.pow(2, 53) is not a safe Integer.");
20      if(Number.isSafeInteger(Math.pow(2, 53) - 1))
21          console.log("Math.pow(2, 53) - 1 is a safe Integer.");
22      else
23          console.log("Math.pow(2, 53) - 1 is not a safe Integer.");
24  </script>
```

关于【代码 3-20】的分析如下：

第 04 行和第 10 行代码在浏览器控制台中输出了 Number.MAX_SAFE_INTEGER 和 Number.MIN_SAFE_INTEGER 这两个安全整数值的内容。

第 05～08 行和第 11～14 行代码通过 if 条件判断语句分别测试了 Number.MAX_SAFE_INTEGER 和 Number.MIN_SAFE_INTEGER 这两个安全整数值与界限值-2^53 和 2^53 的逻辑关系。

第 16～19 行和第 20～23 行代码分别测试了如何使用新增的 Number.isSafeInteger()函数方法判断某个整数值是否在安全整数值范围内。

运行页面，调试信息如图 3.20 所示。

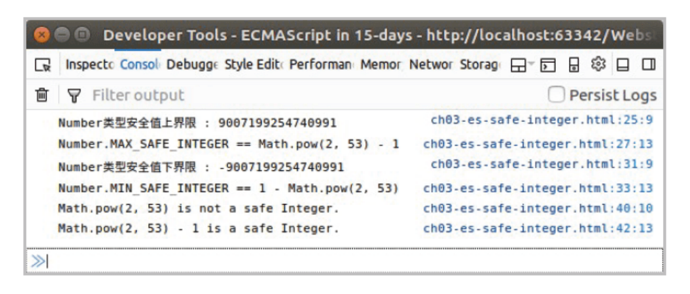


图 3.20　Number 类型特殊值（安全整数值）

从图 3.20 中可以看到，浏览器控制台中输出了安全整数值 Number.MAX_SAFE_INTEGER 和

Number.MIN_SAFE_INTEGER 的具体值。同时，通过安全整数值 Number.MAX_SAFE_INTEGER 和 Number.MIN_SAFE_INTEGER 与界限值-2^53 和 2^53 的逻辑判断结果，可以得出 Number.MAX_SAFE_INTEGER 和 Number.MIN_SAFE_INTEGER 与界限值-2^53 和 2^53 的关系。

另外，通过使用 Number.isSafeInteger()函数方法可以判断出某个整数值是否在安全整数值 Number.MAX_SAFE_INTEGER 和 Number.MIN_SAFE_INTEGER 范围内。

3.6.5　Number.EPSILON

本小节将介绍一个 ECMAScript 6 语法规范中新增的特殊值——Number.EPSILON。EPSILON 用于描述极小的常量，目的是为浮点数运算设置一个误差范围，因为浮点数运算是不精确的。

看一下 Number.EPSILON 的代码示例（详见源代码 ch03 目录中的 ch03-es-number-EPSILON.html 文件）。

【代码 3-21】

```
01  <script type="text/javascript">
02       "use strict";
03       console.log("Number.EPSILON = " + Number.EPSILON);
04     console.log("Number.EPSILON = " + Number.EPSILON.toFixed(16));
05     console.log("0.1 + 0.2 = " + (0.1 + 0.2));
06     console.log("0.1 + 0.2 - 0.3 = " + (0.1 + 0.2 - 0.3));
07     console.log("0.1 + 0.2 - 0.3 = " + (0.1 + 0.2 - 0.3).toFixed(16));
08  </script>
```

关于【代码 3-21】的分析如下：

第 03 行代码尝试输出 Number.EPSILON 极小常量的具体值。

第 04 行代码尝试输出将 Number.EPSILON 极小常量转换为固定 16 位的小数后的具体值。

第 05～07 行代码进行了一个简单的浮点数运算测试。

运行页面，调试信息如图 3.21 所示。从第 05～07 行代码输出的结果可以证明浮点数的运算存在不精确的问题。

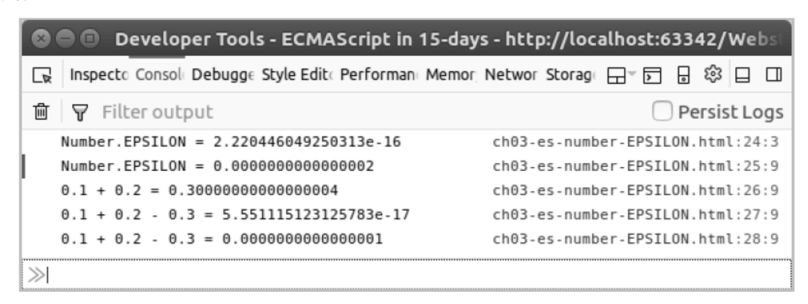


图 3.21　Number.EPSILON（1）

因此，ECMAScript 6 语法规范中引入 Number.EPSILON 特殊值的目的，就是为了验证浮点运算结果的精确性。如果浮点运算结果的误差小于 Number.EPSILON，我们就认为运算结果是正确的。

看一下通过 Number.EPSILON 验证浮点运算结果是否正确的代码示例（详见源代码 ch03 目录中的 ch03-es-number-EPSILON-validate.html 文件）。

【代码 3-22】

```
01  <script type="text/javascript">
02      "use strict";
03     console.log("0.1 + 0.2 - 0.3 = " + (0.1 + 0.2 - 0.3));
04      if((0.1 + 0.2 - 0.3) < Number.EPSILON)
05          console.log("0.1 + 0.2 - 0.3 = " + (0.1 + 0.2 - 0.3) + " is correct.");
06      else
07          console.log("0.1 + 0.2 - 0.3 = " + (0.1 + 0.2 - 0.3) + " is not correct.");
08  </script>
```

关于【代码 3-22】的分析如下：

这段代码主要通过将浮点运算结果的误差值与 Number.EPSILON 进行比较，若小于则证明浮点运算结果是正确的。

运行页面，调试信息如图 3.22 所示。从第 05 行代码输出的结果可以证明浮点运算结果（0.1 + 0.2 - 0.3 = 5.551115123125783e-17）是正确的。

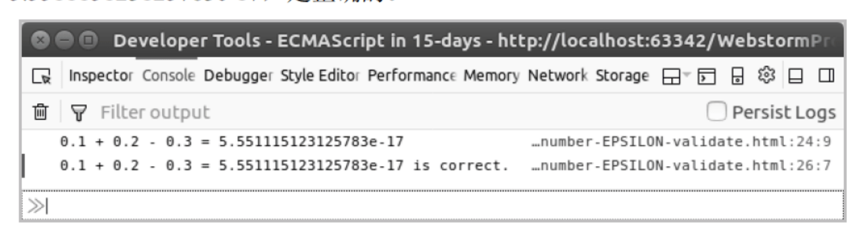


图 3.22　Number.EPSILON（2）

3.7　String 原始类型

本节将介绍 ECMAScript 语法规范中的第 5 种原始类型——String。String 类型同样也是 ECMAScript 5 种原始类型中比较重要的一种。

3.7.1　String 原始类型介绍

从通俗意义上来说，String 类型也就是字符串类型。String 与前几种原始类型的区别在于，它是唯一一种没有固定大小的原始类型，也就是说可以用字符串存储 0 或更多的 Unicode 字符（Unicode 是一种国际通用字符集标准，又称统一字符编码）。

String 类型字符串中的每个字符都有固定的位置，首字符从位置标记 0 开始，第二个字符在位置标记 1 处，以此类推。因此，字符串中的最后一个字符的位置标记一定是字符串的长度减去 1。

3.7.2　定义 String 原始类型

在 ECMAScript 语法规范中，规定 String 类型字符串是通过双引号（"）或单引号（'）来定义声明的。该声明方式与 Java 语言是有区别的，Java 语言必须使用双引号（"）来定义声明字符串，

使用单引号(')定义声明的仅仅是字符。而 ECMAScript 语法规范中恰恰没有定义字符这个概念(字符也被认为是字符串)，所以在定义声明字符串时既可使用双引号（"），也可以使用单引号（'）。

下面看第一个关于定义 String 原始类型的代码示例（详见源代码 ch03 目录中的 ch03-es-string-define.html 文件）。

【代码 3-23】

```
01  <script type="text/javascript">
02      var v_str_a = "Hello ECMAScript!";
03     console.log(v_str_a);
04      var v_str_b = "Hello 'ECMAScript!'";
05     console.log(v_str_b);
06      var v_str_c = 'Hello "ECMAScript!"';
07      console.log(v_str_c);
08  </script>
```

关于【代码 3-23】的分析如下：

第 02、04 和 06 行代码分别通过 var 关键字定义了 3 个变量：v_str_a、v_str_b 和 v_str_c，并分别初始化了不同的字符串。其中，在第 04 行和第 06 行代码初始化的字符串中，演示了如何在定义字符串时以嵌套方式使用双引号（"）和单引号（'）。

运行页面，调试信息如图 3.23 所示。如果我们想输出带有双引号（"）和单引号（'）的字符串，那么在定义字符串时必须以嵌套方式将双引号（"）或单引号（'）加进去。

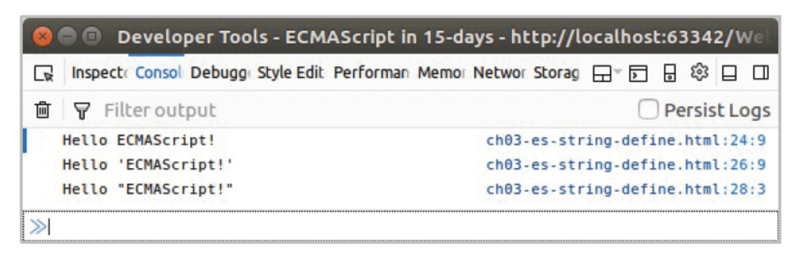


图 3.23　定义 String 原始类型

3.7.3　字符串连接

在 ECMAScript 语法规范中，规定可以通过运算符（+）实现字符串的连接操作，这也是比较简便的字符串连接方式。

看一下字符串连接的代码示例（详见源代码 ch03 目录中的 ch03-es-string-link.html 文件）。

【代码 3-24】

```
01  <script type="text/javascript">
02   var v_str_a = "Hello";
03   var v_str_b = "ECMA";
04   var v_str_c = "Script";
05   console.log(v_str_a + " " + v_str_b + v_str_c + "!");
06  </script>
```

关于【代码 3-24】的分析如下：

这段代码演示了如何进行字符串的连接操作。如果打算输出“Hello ECMAScript!”这样的字

符串，具体做法如下：

第 02～04 行代码通过 var 关键字定义了 3 个变量：v_str_a、v_str_b 和 v_str_c，并分别初始化了字符串（"Hello" "Ecma"和"Script"）。

第 05 行代码通过运算符“+”将 3 个变量（v_str_a、v_str_b 和 v_str_c）进行了连接操作。同时，在第一个和第二个变量之间插入了空格（" "）字符串，在第三个变量后面插入了标点符号字符串（"!"），然后在浏览器控制台窗口中进行了输出。

运行页面，调试信息如图 3.24 所示。使用运算符“+”就可以有效地对字符串进行连接操作，这个操作方式在具体设计中十分有用。

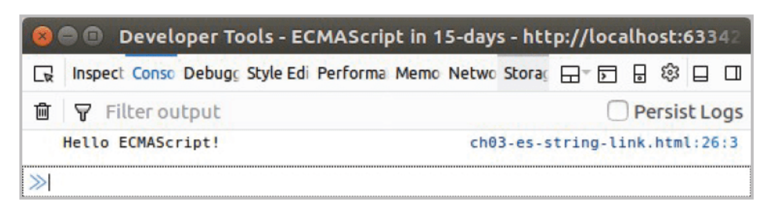


图 3.24　字符串连接

3.7.4　特殊字符串

在 ECMAScript 语法规范中定义了一些特殊的字符串，这些特殊字符串在某些特定环境下非常有用。ECMAScript 语法规范中定义的一些特殊字符串详见表 3.1。

表 3.1　ECMAScript 特殊字符串

编码	描述
\n	换行
\b	空格
\t	制表符
\r	回车
\\	反斜杠
\'	单引号
\"	双引号

看一个使用 String 类型特殊字符串的代码示例（详见源代码 ch03 目录中的 ch03-es-string-spec.html 文件）。

【代码 3-25】

```
01 <script type="text/javascript">
02  var v_str_a = "Hello";
03  var v_str_b = "ECMA";
04  var v_str_c = "Script";
05  console.log(v_str_a + "\b\'\n" + v_str_b + "\"\n\r" + v_str_c + "\t!\\");
06 </script>
```

关于【代码 3-25】的分析如下：

这段代码演示了如何使用 String 类型特殊字符串，第 05 行代码中通过插入多个特殊字符串

（"\b\'\n"、"\"\n\r"、"\t!\\"）进行了字符串的连接操作，然后在浏览器控制台窗口中进行了输出。

运行页面，调试信息如图 3.25 所示。使用特殊字符串就可以实现空格、换行及添加标点符号的效果，这在具体设计中是非常实用的。

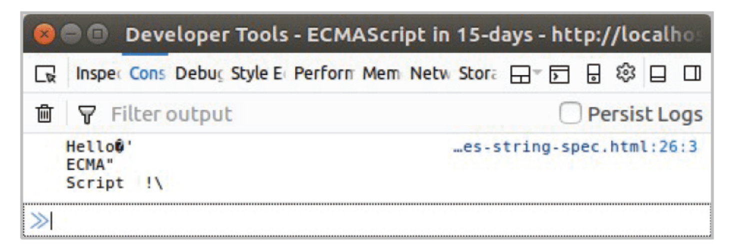


图 3.25　特殊字符串

3.7.5　获取字符串长度

在 ECMAScript 语法规范中，规定通过 String 类型的“length”属性可以获取字符串的长度。

下面看一下获取 String 类型字符串长度的代码示例（详见源代码 ch03 目录中的 ch03-es-string-length.html 文件）。

【代码 3-26】

```
01  <script type="text/javascript">
02   var v_str = "Hello ECMAScript!";
03   console.log(v_str.length);
04   var v_i = 123;
05   console.log(v_i.length);
06   var v_null = null;
07   console.log(v_null.length);
08  </script>
```

关于【代码 3-26】的分析如下：

第 02 行代码通过 var 关键字定义了第一个变量 v_str，并初始化赋值了一个字符串。

第 03 行代码通过“length”属性获取了字符串变量 v_str 的长度。

第 04 行代码通过 var 关键字定义了第二个变量 v_i，并初始化赋值了一个整数数值。

第 05 行代码试图通过“length”属性获取整数变量 v_i 的长度。定义这行代码的目的就是想测试一下“length”属性是否对 Number 类型的变量有效。

第 06 行代码通过 var 关键字定义了第三个变量 v_null，并初始化赋值了 null 原始值。

第 07 行代码试图通过“length”属性获取整数变量 v_null 的长度。同样，定义这行代码的目的也是想测试一下“length”属性是否对 Null 类型的变量有效。

运行页面，调试信息如图 3.26 所示。“length”属性对于字符串是有效的，而对于 Number 类型数值是无效的（即使可以使用“length”属性）。另外，如图 3.26 中箭头所指，对 Null 类型使用“length”属性是会报错的（提示变量类型为 null），这一点需要设计人员在使用“length”属性时加以注意。

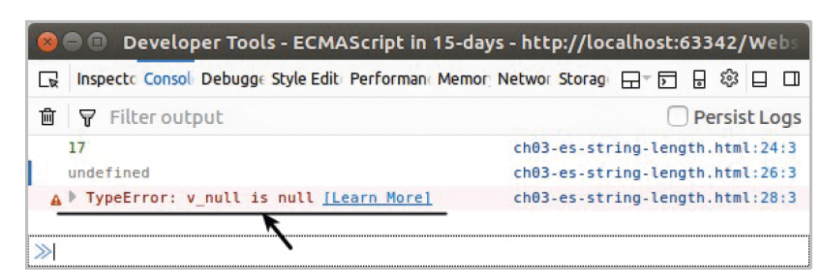


图 3.26　获取字符串长度

3.7.6　字符的 Unicode 编码表示

在 ECMAScript 语法规范中，一般字符内部是通过 Unicode 编码（\u0000～\uFFFF）来表示的。同时，String 类型还定义了 charAt()和 charCodeAt()两个方法来实现对字符的操作。

下面先看一个使用字符 Unicode 编码的代码示例（详见源代码 ch03 目录中的 ch03-es-string-unicode.html 文件）。

【代码 3-27】

```
01  <script type="text/javascript">
02      "use strict";
03      var v_str_unicode = "\u7F16\u7A0B";   // TODO: define unicode String
04     console.log("\\u7F16\\u7A0B : " + v_str_unicode);
05     console.log("\u7F16\u7A0B length : " + v_str_unicode.length);
06     console.log("charAt(0)  : " + v_str_unicode.charAt(0));
07     console.log("charCodeAt(0)  : " + v_str_unicode.charCodeAt(0));
08     console.log("charAt(1)  : " + v_str_unicode.charAt(1));
09     console.log("charCodeAt(1)  : " + v_str_unicode.charCodeAt(1));
10  </script>
```

关于【代码 3-27】的分析如下：

第 03 行代码通过 Unicode 编码方式定义了一个 String 类型变量 v_str_unicode。

第 04 和第 05 行代码输出了字符串变量 v_str_unicode 的汉字内容，通过“length”属性获取了字符串变量 v_str_unicode 的长度。

第 06~09 行代码分别通过 charAt()和 charCodeAt()方法获取了字符串变量 v_str_unicode 的单个字符及其十进制编码。

运行页面，调试信息如图 3.27 所示。通过 Unicode 编码方式定义的字符串变量 v_str_unicode 的内容为汉字“编程”。同时，charAt()方法依次获取了单个汉字字符，charCodeAt()方法获取了汉字字符编码的十进制数值。

早期 ECMAScript 语法规范中的编码仅仅满足于（\u0000～\uFFFF）这个范围，对于识别超出（\uFFFF）范围的编码会出现问题，均是通过两个双字节的变通表示方式来实现的。为了解决上述问题，在 ECMAScript 6 语法规范中新增了通过大括号（{}）形式来识别超出（\uFFFF）范围的编码。

下面再看一个使用 32 位 Unicode 字符编码的代码示例（详见源代码 ch03 目录中的 ch03-es-string-unicode-32.html 文件）。

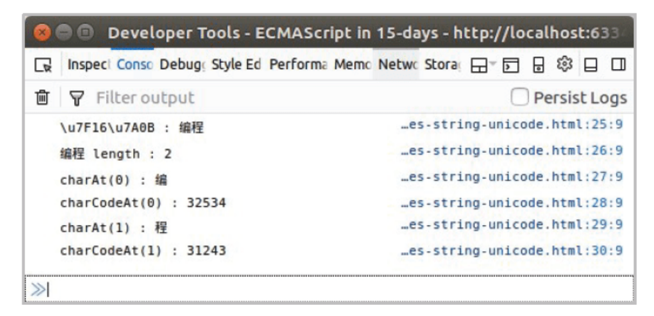


图 3.27　字符 Unicode 表示

【代码 3-28】

```
01  <script type="text/javascript">
02        "use strict";
03        console.log("ECMAScript 打印 UCS-4 字符 :  \u{20BFF}");
04      var v_str_unicode = "\u20BFF";
05      console.log("\\u20BFF : " + v_str_unicode);
06      console.log("charAt(0)  : " + v_str_unicode.charAt(0));
07      console.log("charCodeAt(0)  : " + v_str_unicode.charCodeAt(0).toString(16));
08      console.log("charAt(1)  : " + v_str_unicode.charAt(1));
09      console.log("charCodeAt(1)  : " + v_str_unicode.charCodeAt(1).toString(16));
10  </script>
```

关于【代码 3-28】的分析如下：

第 04 行代码通过 Unicode 编码方式定义了一个 String 类型变量 v_str_unicode，并初始化赋值为“\u20BFF”（本意是想定义一个特殊的汉字“𠯿”）。这里需要特别注意的是，该 Unicode 编码范围超出了（\uFFFF），严格来讲这是一个 32 位 Unicode 字符（官方定义属于 UCS-4 字符，关于 UCS-4 及其相关概念，读者可参考有关编码的文档了解一下）。

第 06～09 行代码分别通过 charAt()和 charCodeAt()方法，获取了字符串变量 v_str_unicode 的单个字符及其十六进制编码。

运行页面，调试信息如图 3.28 所示。如图 3.28 中箭头所指，第 05 行代码输出的字符与预想的完全不同，这是因为字符编码“\u20BFF”无法被旧版本的 ECMAScript 语法规范正确识别（仅能识别 16 位 Unicode 字符），所以第 05 行代码输出了“\u20BF”和“F”的字符组合，而“\u20BF”根本不是一个有效的汉字字符。同时，这个结果通过第 06～09 行代码定义的 charAt()和 charCodeAt()方法也得到了印证。

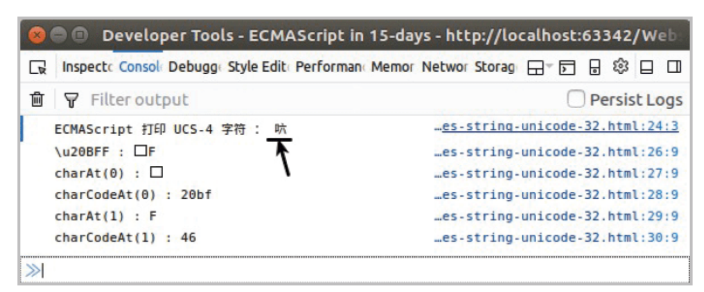


图 3.28　特殊字符 Unicode 表示（1）

那么，旧版本的 ECMAScript 语法规范如何解决【代码 3-28】中出现的无法正确识别字符的问题呢？解决方法就是将 Unicode 编码的 32 位字符转换为 16 位字符。下面继续看一个代码示例（详见源代码 ch03 目录中的 ch03-es-string-unicode-32.html 文件）。

【代码 3-29】

```
01  <script type="text/javascript">
02     "use strict";
03     console.log("ECMAScript 打印 UCS-4 字符 :  \u{20BFF}");
04    var v_str_unicode_16 = "\uD842\uDFFF";
05    console.log("\\uD842\\uDFFF : " + v_str_unicode_16);
06    console.log("charAt(0) : " + v_str_unicode_16.charAt(0));
07    console.log("charCodeAt(0) : " + v_str_unicode_16.charCodeAt(0).toString(16));
08    console.log("charAt(1) : " + v_str_unicode_16.charAt(1));
09    console.log("charCodeAt(1) : " + v_str_unicode_16.charCodeAt(1).toString(16));
10  </script>
```

关于【代码 3-29】的分析如下：

第 04 行代码通过 Unicode 编码方式定义了一个 String 类型变量 v_str_unicode_16，并初始化赋值为“\uD842\uDFFF”（将 32 位编码“\u20BF”转换为了 16 位编码）。

第 06～09 行代码分别通过 charAt()和 charCodeAt()方法获取了字符串变量 v_str_unicode_16 的单个字符及其十六进制编码。

运行页面，调试信息如图 3.29 所示。第 05 行代码输出了正确的汉字字符，这是因为字符编码“\uD842\uDFFF”能够被旧版本的 ECMAScript 语法规范正确识别。

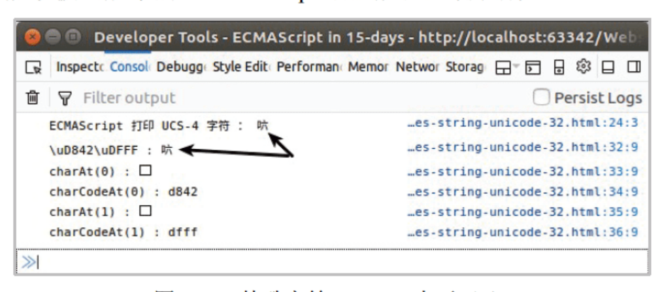


图 3.29　特殊字符 Unicode 表示（2）

目前，ECMAScript 6 语法规范有效地解决了【代码 3-28】中出现的无法正确识别字符的问题。解决方法其实很简单，就是使用大括号（{}）将 32 位的 Unicode 字符编码包含进去。

另外，ECMAScript 6 语法规范中还新增了一个 codePointAt()方法，用于识别 32 位的 Unicode 字符编码，解决了 charAt()和 charCodeAt()方法只能识别 16 位 Unicode 字符编码的问题。

下面看一个在 ECMAScript 6 语法规范下使用 Unicode 字符编码的代码示例（详见源代码 ch03 目录中的 ch03-es-string-unicode-32.html 文件）。

【代码 3-30】

```
01  <script type="text/javascript">
02    "use strict";
03    console.log("ECMAScript 打印 UCS-4 字符 :  \u{20BFF}");
04   var v_str_unicode_32 = "\u{20BFF}";
05   console.log("\\{u20BFF} : " + v_str_unicode_32);
06   console.log("charAt(0) : " + v_str_unicode_32.charAt(0));
```

```
07    console.log("charCodeAt(0) : " + v_str_unicode_32.charCodeAt(0).toString(16));
08    console.log("charAt(1) : " + v_str_unicode_32.charAt(1));
09    console.log("charCodeAt(1) : " + v_str_unicode_32.charCodeAt(1).toString(16));
10    console.log("codePointAt(0) : " + v_str_unicode_32.codePointAt(0).toString(16));
11  </script>
```

关于【代码 3-30】的分析如下：

第 04 行代码通过 Unicode 编码方式定义了一个 String 类型变量 v_str_unicode_32，并初始化赋值为“\u{20BFF}”（使用大括号（{}）将 32 位编码包含进去）。

第 06～09 行代码分别尝试通过 charAt() 和 charCodeAt() 方法获取了字符串变量 v_str_unicode_32 的单个字符及其十六进制编码。

运行页面，调试信息如图 3.30 所示。第 05 行代码输出了正确的汉字字符，这是因为字符编码“\u{20BFF}”能够被 ECMAScript 6 语法规范正确识别。第 06～09 行代码定义的 charAt()和 charCodeAt()方法无法有效识别 32 位的 Unicode 字符编码“\u{20BFF}”，而第 10 行代码定义的 codePointAt()方法能够成功识别“\u{20BFF}”编码。

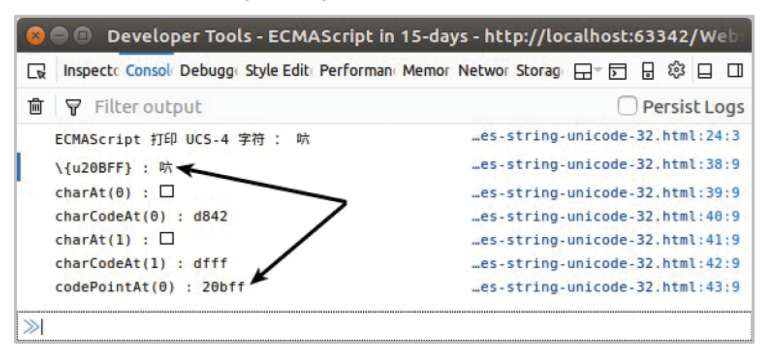


图 3.30　特殊字符 Unicode 表示（3）

在前面几个代码示例中介绍了如何将 Unicode 字符编码进行正确有效的输出。那么能不能直接从十六进制数值编码来获取汉字字符呢？ECMAScript 语法规范中定义了一个 fromCharCode()方法，用于识别 16 位的十六进制数值编码，但该方法无法正确获取 32 位的十六进制数值编码。不过，ECMAScript 6 语法规范中新增了一个 fromCodePoint()方法，用于识别 32 位的十六进制数值编码。

下面看一个直接通过十六进制数值编码来获取汉字字符的代码示例（详见源代码 ch03 目录中的 ch03-es-string-from-unicode.html 文件）。

【代码 3-31】

```
01  <script type="text/javascript">
02      "use strict";
03      console.log("ECMAScript --- String.fromCharCode()方法");
04     console.log("\\u7F16\\u7A0B : " + String.fromCharCode(0x7F16, 0x7A0B));
05     console.log("\\u20BFF : " + String.fromCharCode(0x20BFF));
06     console.log("ECMAScript 6 --- String.fromCodePoint()方法");
07     console.log("\\u20BFF : " + String.fromCodePoint(0x20BFF));
08  </script>
```

关于【代码 3-31】的分析如下：

第 04 行代码通过 fromCharCode()方法将 16 位 Unicode 字符编码"\uD842\uDFFF"转换为汉字字符。

第 05 行代码通过 fromCharCode()方法尝试将 32 位 Unicode 字符编码"\u20BFF"转换为汉字字符。

第 07 行代码通过 fromCodePoint()方法将 32 位 Unicode 字符编码"\u20BFF"转换为汉字字符。

运行页面，调试信息如图 3.31 所示。

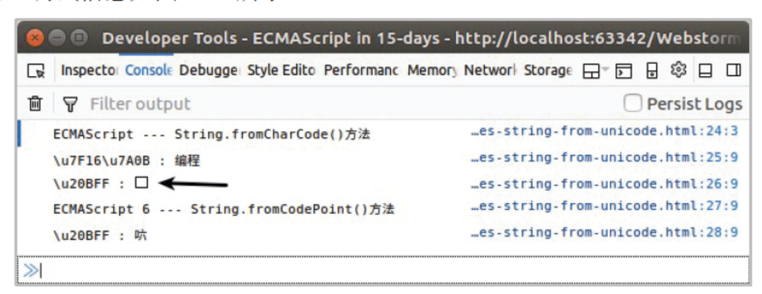


图 3.31　获取特殊汉字字符 Unicode 编码

如图 3.31 中箭头所指，第 04 行代码通过 fromCharCode()方法输出了正确的汉字字符，而第 05 行代码 fromCharCode()方法却无法输出正确的汉字字符，这是因为 fromCharCode()方法无法正确识别 32 位的十六进制数值编码（0x20BFF）。

第 07 行代码通过使用 ECMAScript 6 语法规范中新增的 fromCodePoint()方法正确地将 32 位十六进制数值编码（0x20BFF）转换为了汉字字符。

3.8　本章小结

本章主要介绍了 ECMAScript 语法规范中关于值与类型的知识，包括原始值与引用值、5 种主要原始类型及其新特性等方面的内容，并通过一些具体实例进行了讲解。希望读者通过对本章内容的学习，能够打好 ECMAScript 脚本语言开发的基础。

第 4 章

类型转换

本章将介绍 ECMAScript 类型转换方面的相关知识，特别是 ECMAScript 6 语法规范中关于类型转换的新知识点，这部分内容属于 ECMAScript 脚本语言语法中比较重要的部分。

4.1 转换为字符串

本节将介绍 ECMAScript 语法规范中转换字符串的方法，在 ECMAScript 语法规范中提供了一个 toString(argument)函数方法，用于实现转换成字符串的功能，该方法适用于 Boolean 原始类型、Number 原始类型和 String 原始类型的原始值。

4.1.1 toString()函数方法的语法格式

ECMAScript 语法规范中提供的 toString(argument)函数方法用于实现将 Boolean 原始类型、Number 原始类型和 String 原始类型等数据类型转换为字符串形式。

关于 toString(argument)函数方法的语法说明如下：

```
toString(argument);          // TODO: 用于实现转换成字符串的功能
```

其中，“argument”参数是可选的，当需要转换成特殊类型的字符串时，可以通过定义该参数来实现。

4.1.2 使用默认 toString()函数方法

在应用 toString(argument)函数方法时，常用的是默认不带“argument”参数的形式。

下面看一段使用默认 toString()函数方法实现转换为字符串操作的代码示例（详见源代码 ch04

目录中的 ch04-es-toString-default.html 文件）。

【代码 4-1】

```
01   <script type="text/javascript">
02        var v_str = "toString()";
03        console.log(v_str.toString());
04        var v_int = 123;
05      console.log(v_int.toString());
06      var v_float = 123.123;
07      console.log(v_float.toString());
08        var v_b_t = true;
09        console.log(v_b_t.toString());
10        var v_b_f = false;
11        console.log(v_b_f.toString());
12        var v_null = null;
13        console.log(v_null.toString());
14   </script>
```

关于【代码 4-1】的分析如下：

第 02 行代码定义了一个字符串变量 v_str，并初始化赋值了字符串“toString()”。第 03 行代码应用 toString()函数方法将变量 v_str 转换为字符串，此处读者可能会有一点疑问，既然变量 v_str 已经初始化为字符串了，还可以使用 toString()函数方法再次转换为字符串类型么？答案是肯定的，ECMAScript 语法规范允许这样的操作，因为 String 类型本身是原始类型，定义的变量是伪对象，自然就支持 toString()函数方法。

第 04 行代码定义了一个 Number 类型变量 v_int，并初始化赋值了数值 123。第 05 行代码尝试应用 toString()函数方法将变量 v_int 转换为字符串。

第 06 行代码定义了一个浮点数类型变量 v_float，并初始化赋值了浮点数值 123.123。第 07 行代码尝试应用 toString()函数方法将变量 v_float 转换为字符串。

第 08 行和第 10 行代码分别定义了两个布尔型变量，并分别初始化赋值为 Boolean 类型的原始值 true 和 false。第 09 行和第 11 行代码尝试应用 toString()函数方法将 Boolean 变量转换为字符串。

第 12 行代码定义了一个 Null 类型变量 v_null，并初始化赋值了 Null 类型的原始值（null）。第 13 行代码应用 toString()函数方法将 Null 类型变量 v_null 转换为字符串。定义这两行代码的目的是测试一下 Null 类型是否支持 toString()函数方法的操作。

运行【代码 4-1】所定义的 HTML 页面，并使用调试器查看控制台输出的调试信息，页面效果如图 4.1 所示。

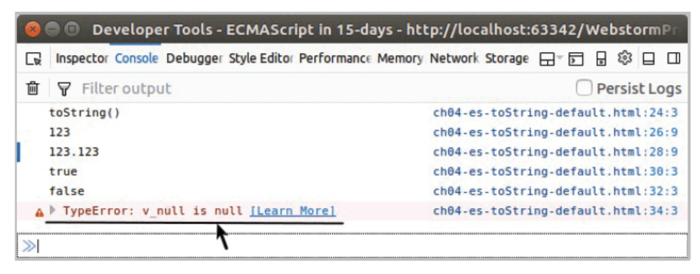


图 4.1　默认 toString()函数方法

如图 4.1 所示，对 String 类型数据应用 toString()函数方法后返回的仍是原字符串的内容，对

Number 类型数据应用 toString()函数方法后返回的是数值的内容，对 Boolean 类型应用 toString()函数方法后返回的是 true 或 false 原始值。

另外，如图 4.1 中箭头所指，如果尝试对 Null 类型应用 toString()函数方法，就会返回类型错误的提示信息，说明 Null 类型是不支持 toString()函数方法的。

4.1.3　Number 类型数值转换为字符串

toString()函数方法常用的应用场景就是将各种 Number 类型数值转换为字符串格式。

下面看一段使用 toString()函数方法将各种 Number 类型数值转换成字符串的代码示例（详见源代码 ch04 目录中的 ch04-es-toString-num.html 文件）。

【代码 4-2】

```
01  <script type="text/javascript">
02       var v_i_1 = 123;
03       console.log("123 toString() = " + v_i_1.toString());
04       var v_i_2 = -123;
05       console.log("-123 toString() = " + v_i_2.toString());
06       var v_f_1 = 123.0;
07       console.log("123.0 toString() = " + v_f_1.toString());
08       var v_f_2 = 123.123;
09       console.log("123.123 toString() = " + v_f_2.toString());
10     var v_oct = 0123;
11     console.log("0123 toString() = " + v_oct.toString());
12     var v_hex = 0x123;
13     console.log("0x123 toString() = " + v_hex.toString());
14       var v_e = 123e6;
15       console.log("123e6 toString() = " + v_e.toString());
16  </script>
```

关于【代码 4-2】的分析如下：

第 02～05 行代码测试了对正整数和负整数应用 toString()函数方法的结果。

第 06～09 行代码测试了对浮点数应用 toString()函数方法的结果。

第 10～11 行代码测试了对八进制数应用 toString()函数方法的结果。

第 12～13 行代码测试了对十六进制数应用 toString()函数方法的结果。

第 14～15 行代码测试了对指数应用 toString()函数方法的结果。

运行页面，调试信息如图 4.2 所示。

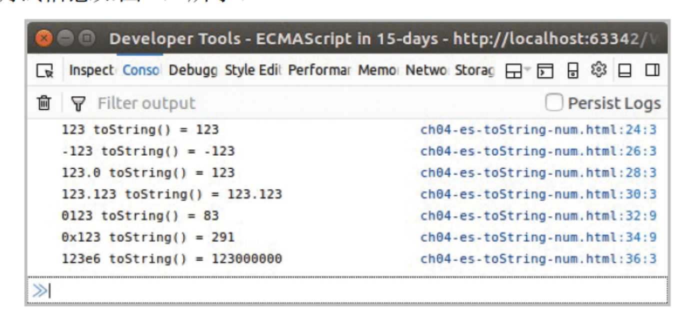


图 4.2　Number 类型数值转换为字符串

从图 4.2 中可以看到，对负整数应用 toString()函数方法，负号（-）也会被当作字符串来处理。

对浮点数应用 toString()函数方法，返回的仍是原浮点数的内容（浮点数 123.0 会自动省略小数点后的 0）。

对八进制和十六进制数值应用 toString()函数方法，会先转换为十进制数值，然后转换为字符串。

对使用科学计数法的数值应用 toString()函数方法，会先转换为完整长度的数值，然后转换为字符串。

4.1.4 使用带参数的 toString()函数方法

toString()函数方法还有一种带参数的使用方式，通过定义“argument”参数可以实现将数值转换为相应进制数的字符串形式。

下面看一段使用 toString(argument)函数方法将 Number 类型数值分别转换成为二进制、八进制和十六进制字符串的代码示例（详见源代码 ch04 目录中的 ch04-es-toString-argument.html 文件）。

【代码 4-3】

```
01  <script type="text/javascript">
02      "use strict";
03      var v_i_2 = 15;
04      console.log("十进制数值 : 15.toString(2) = " + v_i_2.toString(2));
05      var v_b_2 = 0b1111;
06      console.log("二进制数值 : 0b1111.toString(10) = " + v_b_2.toString(10));
07      var v_i_8 = 63;
08      console.log("十进制数值 : 63.toString(8) = " + v_i_8.toString(8));
09      var v_o_8 = 0o77;
10      console.log("八进制数值 : 0o77.toString(10) = " + v_o_8.toString(10));
11      var v_i_16 = 255;
12      console.log("十进制数值 : 255.toString(16) = " + v_i_16.toString(16));
13      var v_h_16 = 0xff;
14      console.log("十六进制数值 : 0xff.toString(10) = " + v_h_16.toString(10));
15  </script>
```

关于【代码 4-3】的分析如下：

第 03 行代码定义了第一个变量 v_i_2，并初始化赋值了十进制数值 15。

第 04 行代码应用 toString(2)函数方法将变量 v_i_2 转换为二进制字符串，其中参数“2”表示转换为二进制字符串。

第 05 行代码定义了第二个变量 v_b_2，并初始化赋值了二进制数值 0b1111。

第 06 行代码应用 toString(10)函数方法将变量 v_b_2 转换为字符串，其中参数“10”表示转换为十进制字符串。

第 07 行代码定义了第三个变量 v_i_8，并初始化赋值了十进制数值 63。

第 08 行代码应用 toString(8)函数方法将变量 v_i_8 转换为八进制字符串，其中参数“8”表示转换为八进制字符串。

第 09 行代码定义了第四个变量 v_o_8，并初始化赋值了八进制数值 0o77。

第 10 行代码应用 toString(10)函数方法将变量 v_o_8 转换为字符串，其中参数“10”表示转换为十进制字符串。

第 11 行代码定义了第五个变量 v_i_16，并初始化赋值了十进制数值 255。

第 12 行代码应用 toString(16)函数方法将变量 v_i_16 转换为十六进制字符串，其中参数“16”表示转换为十六进制字符串。

第 13 行代码定义了第六个变量 v_h_16，并初始化赋值了十六进制数值 0xff。

第 14 行代码应用 toString(10)函数方法将变量 v_h_16 转换为十进制字符串，其中参数“10”表示转换为十进制字符串。

运行页面，调试信息如图 4.3 所示。

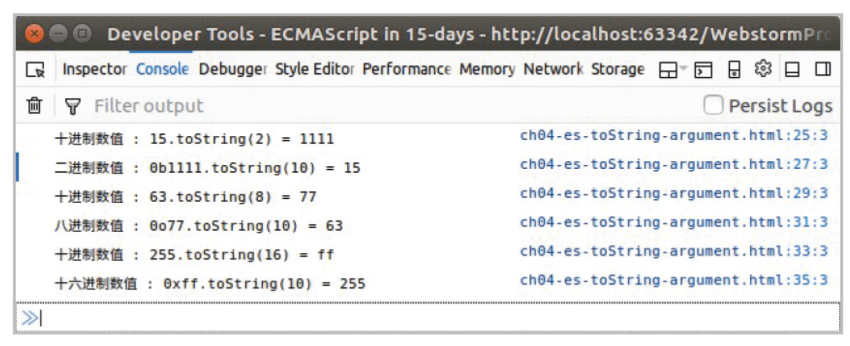


图 4.3 带参数的 toString()方法

从图 4.3 中可以看到，十进制数值 15 对应的二进制数值正是 1111，十进制数值 63 对应的八进制数值正是 0o77，十进制数值 255 对应的十六进制数值正是 0xff。

另外，对于默认的 toString()函数方法，其实是与 toString(10)函数方法一致的，只不过十进制参数 10 可以省略。

4.2 转换为数值

本节将介绍 ECMAScript 语法规范中，把非数字的原始值转换为数值的两种方法：parseInt()和 parseFloat()。parseInt()方法用于将值转换为整数，parseFloat()方法用于将值转换为浮点数。

4.2.1 parseInt()函数方法的语法格式

ECMAScript 语法规范中定义了 parseInt()函数方法，用于实现将 String 原始类型的原始值转换为整数形式。

关于 parseInt()函数方法的语法说明如下：

```
parseInt(strArg, [baseArg]);            // TODO: 用于实现转换为整数数值的功能
```

“strArg”参数是必须的，表示用于转换的 String 类型原始值；“baseArg”参数是可选的，用于指定“strArg”参数的基数（进制数）。

另外，parseInt()函数方法在执行转换操作前，均会分析判断“strArg”参数字符串的格式。具

体方法是依次查看“strArg”参数每个位置处的字符，判断其是否是一个有效数字，直到最后一个位置的字符，parseInt()函数方法将把“strArg”参数字符串转换为数字。即使某个位置出现非有效数字，parseInt()方法也会将该字符之前的字符串转换为数字。但如果首字符为非有效数字字符，那么 parseInt()方法会返回 NaN，并不再继续执行后续操作。

4.2.2 转换为整数数值

ECMAScript 语法规范中的 parseInt()函数方法用于实现将 String 类型原始值转换为整数类型数值，而对于一些非 String 类型原始值的操作会返回 NaN。

下面看一个使用 parseInt()函数方法将字符串转换为数字的代码示例（详见源代码 ch04 目录中的 ch04-es-parseInt.html 文件）。

【代码 4-4】

```
01  <script type="text/javascript">
02       "use strict";
03       var v_i = parseInt("123");
04       console.log("parseInt(\"123\") = " + v_i);
05       var v_f = parseInt("123.123");
06       console.log("parseInt(\"123.123\") = " + v_f);
07       var v_hex = parseInt("0xff");
08       console.log("parseInt(\"0xff\") = " + v_hex);
09       var v_i_str = parseInt("123abc");
10       console.log("parseInt(\"123abc\") = " + v_i_str);
11       var v_str_i = parseInt("str123");
12       console.log("parseInt(\"str123\") = " + v_str_i);
13     var v_t = parseInt(true);
14     console.log("parseInt(\"true\") = " + v_t);
15  </script>
```

关于【代码 4-4】的分析如下：

第 03 行代码定义了第一个变量 v_i，并初始化赋值了通过 parseInt("123")方法将整数字符串转换为数字的返回值。

第 05 行代码定义了第二个变量 v_f，并初始化赋值了通过 parseInt("123.123")方法将浮点数字符串转换为数字的返回值。

第 07 行代码定义了第三个变量 v_hex，并初始化赋值了通过 parseInt("0xff")方法将十六进制字符串转换为数字的返回值。

第 09 行代码定义了第四个变量 v_i_str，并初始化赋值了通过 parseInt("123abc")方法将字符串（"123abc"）转换为数字的返回值。

第 11 行代码定义了第五个变量 v_str_i，并初始化赋值了通过 parseInt("str123")方法将字符串（"str123"）转换为数字的返回值。

第 13 行代码定义了第六个变量 v_t，并初始化赋值了通过 parseInt(true)方法将布尔值（true）转换为数字的返回值。

运行页面，调试信息如图 4.4 所示。

图 4.4　parseInt()函数方法转换为整数

从图 4.4 中可以看到，第 04 行和第 06 行代码输出的结果表明，对于应用 parseInt()函数方法进行转换的字符串，如果是数值类型的，无论是整数类型还是浮点数类型的字符串，返回的均是整数类型。因为 parseInt()函数方法不能识别小数点（.），所以 parseInt()函数方法会自动舍弃浮点数中小数点后面（包含小数点）的数值。

第 08 行代码输出的结果表明，如果字符串定义成十六进制形式的字符串，parseInt()函数方法是能够识别出来的，并按照十六进制数值换算成十进制数值进行返回。

第 10 行代码输出的结果表明，如果字符串定义成类似“123abc”字符串的形式，parseInt()函数方法是能够识别出来前面数值的，并将识别出来的数值进行返回。这是因为 parseInt()函数方法按照从前至后的顺序依次识别字符串中的字符，一旦遇到字符串中字符为非数字，就会马上终止执行，并将已经识别的数字（本例为 123）进行返回。

第 12 行代码输出的结果表明，如果字符串定义成类似“str123”字符串的形式，parseInt()函数方法会认为是无效字符串，并返回原始值 NaN。

第 14 行代码输出的结果表明，parseInt()函数方法是无法操作布尔类型值的，对于布尔值会返回原始值 NaN。

4.2.3　转换指定基数的整数数值

ECMAScript 语法规范中的 parseInt()函数方法还有一种基数方式，可以用于实现将二进制、八进制、十六进制或其他任何进制的 String 类型原始值转换为整数类型数值。而基数则是通过 parseInt()函数方法的第二个参数 baseArg 来指定的。

下面看一个使用 parseInt()函数方法基数方式转换为数字的代码示例（详见源代码 ch04 目录中的 ch04-es-parseInt-baseArg.html 文件）。

【代码 4-5】

```
01  <script type="text/javascript">
02      "use strict";
03      var v_i_2 = 11;
04      console.log("parseInt(11, 2) = " + parseInt(v_i_2, 2));
05      var v_i_8 = 77;
06      console.log("parseInt(77, 8) = " + parseInt(v_i_8, 8));
07      var v_i_16 = "ff";
08      console.log("parseInt(ff, 16) = " + parseInt(v_i_16, 16));
09  </script>
```

关于【代码 4-5】的分析如下：

这段代码使用了带参数的 parseInt()函数方法，可以将字符串转换为按照参数（2、8、16）定义的二进制、八进制和十六进制数。

第 03 和第 04 行代码定义了第一个变量 v_i_2，并初始化赋值了数值 11，并通过 parseInt(v_i_2, 2)方法将数值 11 指定为二进制数。

第 05 和第 06 行代码定义了第二个变量 v_i_8，并初始化赋值了数值 77，并通过 parseInt(v_i_8, 8)方法将数值 77 指定为八进制数。

第 07 和第 08 行代码定义了第三个变量 v_i_16，初始化赋值了字符串"ff"，并通过 parseInt(v_i_16, 16)方法将字符串"ff"指定为十六进制数。

运行页面，调试信息如图 4.5 所示。使用了基数方式的 parseInt()函数方法，可以将第一个参数定义的目标值按照第二个参数指定的进制进行转换。

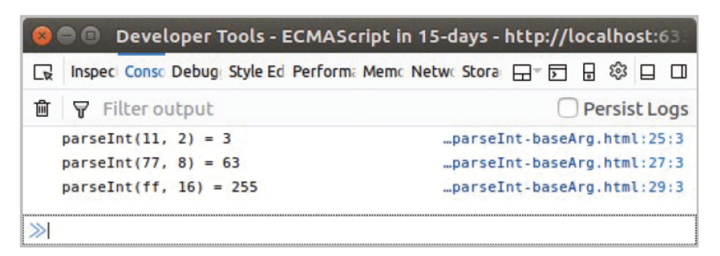


图 4.5　parseInt()方法基数方式

另外，对于默认的 parseInt()函数方法，其实是省略了第二个参数（指定为十进制）的，也就是说 parseInt()函数方法默认是按照转换为十进制数来操作的。

4.2.4　parseFloat()函数方法的语法格式

ECMAScript 语法规范中定义了 parseFloat()函数方法，用于实现将 String 原始类型的原始值转换为浮点数形式。

关于 parseFloat()函数方法的语法说明如下：

```
parseFloat(strArg);            // TODO: 用于实现转换为浮点数的功能
```

其中，“strArg”参数是必需的，表示用于转换的 String 类型原始值。

parseFloat()方法与 parseInt()方法的处理方式相似，从首位置字符开始依次查看每个字符，直到找到第一个非有效的字符为止，然后把该字符之前的字符串转换为浮点数（或整数）。另外，parseFloat()方法也没有类似 parseInt()方法的基数方式。

对于 parseFloat()方法来说有一点特殊之处，就是第一个出现的小数点是有效字符（可用于实现浮点数的识别）。但如果有两个小数点，那么第二个小数点将被看作是无效的，此时 parseFloat()函数方法会把这个小数点之前的字符转换为数字，如字符串“123.456.789”就会被解析成浮点数 123.456。

同时，使用 parseFloat()方法的另一点特殊之处在于，字符串必须以十进制形式表示浮点数。因为该方法会忽略前导数值 0，所以八进制数 063 将被解析为十进制数 63，而严格模式下的八进制数 0o63 将被解析为 NaN。同理，对于十六进制数 0xff，也将被解析返回 NaN，因为在浮点数中，

字符“o”和“x”并不是有效的字符（不过，在实际测试过程中会发现，大部分主流浏览器会解析返回数值 0，而不是语法规范所定义的 NaN）。

4.2.5　转换为浮点数

在 ECMAScript 语法规范中，parseFloat()函数方法用于实现将 String 类型原始值转换为浮点数形式。但是，根据 String 原始值的不同格式，也不一定必须解析为浮点数格式，还可以解析为整数格式。

下面看一个使用 parseFloat()函数方法转换为浮点数的代码示例（详见源代码 ch04 目录中的 ch04-es-parseFloat.html 文件）。

【代码 4-6】

```
01  <script type="text/javascript">
02      "use strict";
03     var v_i = parseFloat("123");
04     console.log('parseFloat("123") = ' + v_i);
05      var v_f_1 = parseFloat("123.0");
06      console.log('parseFloat("123.0") = ' + v_f_1);
07      var v_f_3 = parseFloat("123.123");
08      console.log('parseFloat("123.123") = ' + v_f_3);
09      var v_f_f = parseFloat("123.456.789");
10      console.log('parseFloat("123.456.789") = ' + v_f_f);
11     var v_f_oct = parseFloat("063");
12     console.log('parseFloat("063") = ' + v_f_oct);
13      var v_f_o_oct = parseFloat("0o63");
14      console.log('parseFloat("0o63") = ' + v_f_o_oct);
15      var v_f_hex = parseFloat("0xff");
16      console.log('parseFloat("0xff") = ' + v_f_hex);
17      var v_f_str = parseFloat("123.abc");
18      console.log('parseFloat("123.abc") = ' + v_f_str);
19      var v_str_f = parseFloat("abc.123");
20      console.log('parseFloat("abc.123") = ' + v_str_f);
21  </script>
```

关于【代码 4-6】的分析如下：

第 03 行代码定义了第一个变量 v_i，并初始化赋值了通过 parseFloat("123")方法将整数字符串"123"转换为浮点数返回。

第 05 行代码定义了第二个变量 v_f_1，并初始化赋值了通过 parseFloat("123.0")方法将一位小数（数字 0）的浮点数字符串转换为浮点数返回。

第 07 行代码定义了第三个变量 v_f_3，并初始化赋值了通过 parseFloat("123.123")方法将三位小数的浮点数字符串转换为浮点数返回。

第 09 行代码定义了第四个变量 v_f_f，并初始化赋值了通过 parseFloat("123.456.789")方法将含有两个小数点的字符串转换为浮点数返回。

第 11 行代码定义了第五个变量 v_f_oct，并初始化赋值了通过 parseFloat("063")方法将八进制格式的数字字符串转换为浮点数返回。

第 13 行代码定义了第六个变量 v_f_o_oct，并初始化赋值了通过 parseFloat("0o63")方法将八进

制格式的数字字符串转换为浮点数返回。注意，这里的八进制数字字符串 0o63 为严格模式下的八进制数值定义方式。

第 15 行代码定义了第七个变量 v_f_hex，并初始化赋值了通过 parseFloat("0xff")方法将十六进制格式的浮点数字符串转换为浮点数返回。

第 17 行代码定义了第八个变量 v_f_str，并初始化赋值了通过 parseFloat("123.abc")方法将字符串转换为浮点数返回。这里使用字符串“123.abc”的目的是想测试一下 parseFloat()函数方法如何识别非数字字符串。

第 19 行代码定义了第九个变量 v_str_f，并初始化赋值了通过 parseFloat("abc.123")方法将字符串转换为浮点数返回。这里使用字符串“abc.123”的目的同样是想测试一下 parseFloat()函数方法如何识别非数字字符串。

运行页面，调试信息如图 4.6 所示。

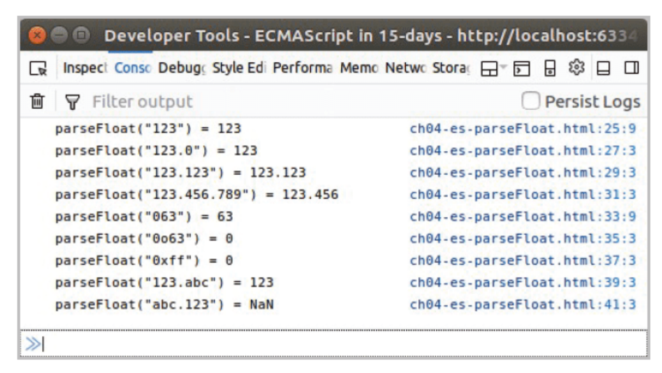


图 4.6　parseFloat()函数方法

从图 4.6 中可以看到，第 04 行代码输出的结果表明，对于应用 parseFloat()函数方法的整数字符串，同样会返回整数。

第 06 行代码输出的结果表明，对于应用 parseFloat()函数方法的浮点数字符串（但小数部分为 0），会舍去小数部分返回整数。

第 08 行代码输出的结果表明，对于应用 parseFloat()函数方法的浮点数字符串，同样会返回浮点数。

第 10 行代码输出的结果表明，如果字符串定义成含有多个小数点（.）字符串（"123.456.789"），那么 parseFloat()函数方法仍可以返回一个有效浮点数（123.456），只不过会舍弃第二个小数点后面的数字。这是因为 parseFloat()函数方法会将第二个小数点（.）后面的字符串（包含第二个小数点）全部视为无效字符。

第 12 行代码输出的结果表明，将字符串定义成八进制形式的字符串（063）是没有意义的，因为 parseFloat()函数方法只会将字符串按照十进制数值来处理，所以得到的结果是数值 63。

第 14 行代码输出的结果表明，将字符串定义成严格模式下的八进制形式的字符串（0o63）也是没有意义的，因为 parseFloat()函数方法只会将字符“o”按照无效字符来处理，所以得到的结果是 0。

第 16 行代码输出的结果表明，将字符串定义成十六进制形式的字符串（0xff）也是没有意义的，因为 parseFloat()方法只会将字符“x”按照无效字符来处理，所以得到的结果同样是 0。

第 18 行代码输出的结果表明，将字符串定义成类似“123.abc”字符串的形式，parseFloat()函

4.3 强制类型转换

4.3.1 强制类型转换基础

4.3.2 强制转换为 Number 类型

【代码 4-7】

```
06        console.log('Number("123.456") = ' + Number("123.456"));
07        console.log('Number("123.456.789") = ' + Number("123.456.789"));
08        console.log("Number(true) = " + Number(true));
09        console.log("Number(false) = " + Number(false));
10        console.log("Number(null) = " + Number(null));
11        console.log("Number(undefined) = " + Number(undefined));
12    </script>
```

关于【代码 4-7】的分析如下：

第 02 行代码通过 Number(123)方法将整数 123 强制类型转换为 Number 类型。

第 03 行代码通过 Number("123")方法将字符串“123”强制类型转换为 Number 类型。

第 04 行代码通过 Number("abc")方法将字符串“abc”强制类型转换为 Number 类型。

第 05 行代码通过 Number(123.456)方法将浮点数 123.456 强制类型转换为 Number 类型。

第 06 行代码通过 Number("123.456")方法将字符串“123.456”强制类型转换为 Number 类型。

第 07 行代码通过 Number("123.456.789")方法将字符串“123.456.789”强制类型转换为 Number 类型。

第 08 和第 09 行代码通过 Number(true)和 Number(false)方法将 Boolean 类型值强制类型转换为 Number 类型。

第 10 行代码通过 Number(null)方法尝试将 Null 类型值强制类型转换为 Number 类型。

第 11 行代码通过 Number(undefined)方法尝试将 Undefined 类型值强制类型转换为 Number 类型。

运行页面，调试信息如图 4.7 所示。

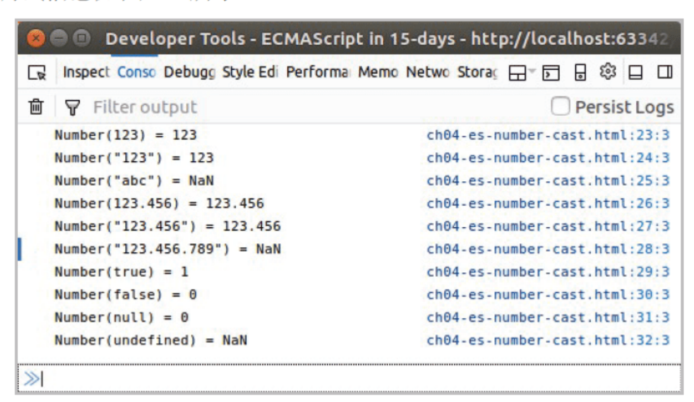


图 4.7　Number()强制类型转换

从图 4.7 中可以看到，第 02 和第 03 行代码输出的结果表明，对数值或数值类型的字符串使用 Number()方法均会强制类型转换为数字并返回。

第 04 行代码输出的结果表明，对非数值类型的字符串使用 Number()函数方法会被认为是无效的，并返回原始值 NaN。

第 05 行和第 06 行代码输出的结果表明，对浮点数或浮点数类型的字符串使用 Number()函数方法均会强制类型转换为数字并返回。

第 07 行代码输出的结果表明，对类似“123.456.789”的字符串使用 Number()函数方法会被认为是无效的，并返回原始值 NaN。

第 08 和第 09 行代码输出的结果表明，对 Boolean 类型值（true 和 false）使用 Number()函数

方法会强制类型转换为数字并返回。true 返回值 1，flase 返回值 0，这点与其他高级编程语言是一致的。

第 10 行代码输出的结果表明，对 Null 类型值（null）使用 Number()函数方法会被认是有效的，并返回数值 0。

第 11 行代码输出的结果表明，对 Undefined 类型值（undefined）使用 Number()函数方法会被认为是无效的，并返回原始值 NaN。

以上就是使用 Number()函数方法对大部分类型进行强制类型转换的结果，读者在使用 Number()函数方法时需要加以注意。

4.3.3　强制转换为 Boolean 类型

在 ECMAScript 语法规范中，Boolean()函数方法可以把给定的值（布尔值、整数值、字符串、特殊原始值等）转换为布尔类型的值。

下面看一个使用 Boolean()函数方法进行强制类型转换的代码示例（详见源代码 ch04 目录中的 ch04-es-boolean-cast.html 文件）。

【代码 4-8】

```
01  <script type="text/javascript">
02      console.log("Boolean(true) = " + Boolean(true));
03      console.log("Boolean(false) = " + Boolean(false));
04      console.log("Boolean(1) = " + Boolean(1));
05      console.log("Boolean(10) = " + Boolean(10));
06      console.log("Boolean(0) = " + Boolean(0));
07      console.log('Boolean("abc") = ' + Boolean("abc"));
08      console.log('Boolean("") = ' + Boolean(""));
09      console.log("Boolean(null) = " + Boolean(null));
10      console.log("Boolean(undefined) = " + Boolean(undefined));
11  </script>
```

关于【代码 4-8】的分析如下：

第 02 行代码通过 Boolean(true)方法将 true 强制类型转换为 Boolean 类型。

第 03 行代码通过 Boolean(false)方法将 false 强制类型转换为 Boolean 类型。

第 04 行代码通过 Boolean(1)方法将数值 1 强制类型转换为 Boolean 类型。

第 05 行代码通过 Boolean(10)方法将数值 10 强制类型转换为 Boolean 类型。

第 06 行代码通过 Boolean(0)方法将数值 0 强制类型转换为 Boolean 类型。

第 07 行代码通过 Boolean("abc")方法将字符串“abc”强制类型转换为 Boolean 类型。

第 08 行代码通过 Boolean("")方法将空字符串强制类型转换为 Boolean 类型。

第 09 行代码通过 Boolean(null)方法尝试将 Null 类型值强制类型转换为 Boolean 类型。

第 10 行代码通过 Boolean(undefined)方法尝试将 Undefined 类型值强制类型转换为 Boolean 类型。

运行页面，调试信息如图 4.8 所示。

从图 4.8 中可以看到，第 02 和第 03 行代码输出的结果表明，对 Boolean 类型值使用 Boolean()函数方法进行强制类型转换，会得到 Boolean 原始值（true 或 false）的返回值。

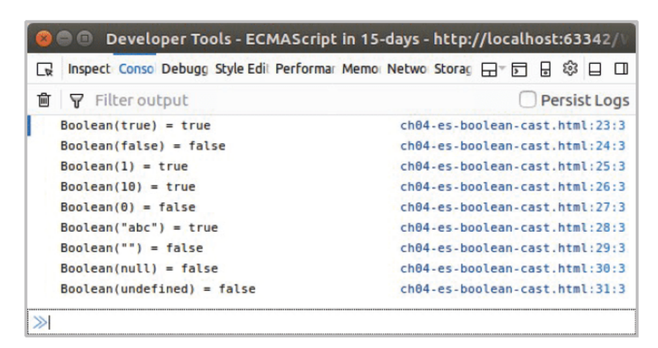


图 4.8　Boolean()强制类型转换

第 04 和第 05 行代码输出的结果表明，对非零数值使用 Boolean()函数方法进行强制类型转换，会得到 Boolean 原始值（true）的返回值。

第 06 行代码输出的结果表明，对数值 0 使用 Boolean()函数方法进行强制类型转换，会得到 Boolean 原始值（false）的返回值。

第 07 行代码输出的结果表明，对非空的字符串使用 Boolean()函数方法进行强制类型转换，会得到 Boolean 原始值（true）的返回值。

第 08 行代码输出的结果表明，对非空的字符串使用 Boolean()函数方法进行强制类型转换，会得到 Boolean 原始值（false）的返回值。

第 09 行代码输出的结果表明，对 Null 类型值（null）使用 Boolean()函数方法进行强制类型转换，会得到 Boolean 原始值（false）的返回值。

第 10 行代码输出的结果表明，对 Undefined 类型值（undefined）使用 Boolean()函数方法进行强制类型转换，会得到 Boolean 原始值（false）的返回值。

以上就是使用 Boolean()函数方法对大部分类型进行强制类型转换的结果，读者在使用 Boolean()函数方法时需要加以注意。

4.3.4　强制转换为 String 类型

在 ECMAScript 语法规范中，String()函数方法可以把给定的值（布尔值、整数值、字符串、特殊原始值等）转换为 String 类型的值。

下面看一个使用 String()函数方法进行强制类型转换的代码示例（详见源代码 ch04 目录中的 ch04-es-string-cast.html 文件）。

【代码 4-9】

```
01  <script type="text/javascript">
02      var v_num = 123;
03      console.log("String(123) = " + String(v_num));
04     var v_t = true;
05     console.log("String(true) = " + String(v_t));
06      var v_str_null = String(null);
07      console.log("String(null) = " + v_str_null);
08      var v_str_undefined = String(undefined);
09      console.log("String(undefined) = " + v_str_undefined);
```

```
10      var v_null = null;
11      console.log("null.toString() = " + v_null.toString());
12       var v_undefined;
13       console.log("undefined.toString() = " + v_undefined.toString());
14   </script>
```

关于【代码 4-9】的分析如下：

第 02 行代码定义了第一个变量 v_num，并初始化赋值了数字 123。

第 03 行代码通过 String(123)方法将数字 123 强制类型转换为 String 类型值。

第 04 行代码定义了第二个变量 v_t，并初始化赋值了布尔值 true。

第 05 行代码通过 String(true)方法将布尔值 true 强制类型转换为 String 类型值。

第 06 行代码定义了第三个变量 v_str_null，并初始化赋值了通过 String(null)方法将原始值 null 强制类型转换为 String 类型值。

第 08 行代码定义了第四个变量 v_str_undefined，并初始化赋值了通过 String(undefined)方法将原始值 undefined 强制类型转换为 String 类型值。

第 10 行代码定义了第五个变量 v_null，并初始化赋值了原始值 null。

第 11 行代码尝试通过 toString()方法将变量 v_null 转换为字符串格式。

第 12 行代码定义了第六个变量 v_undefined，并未进行初始化操作。

第 13 行代码尝试通过 toString()方法将变量 v_undefined 转换为字符串格式。

运行页面，调试信息如图 4.9 所示。

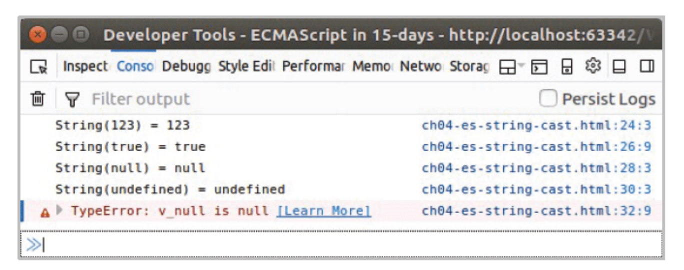


图 4.9　String()强制类型转换（1）

从图 4.9 中可以看到，对整数值和布尔值使用 String()函数方法进行强制类型转换时，可以正确生成字符串而不会引发错误。

同样，对原始值 null 或 undefined 使用 String()函数方法进行强制类型转换时，可以正确生成字符串而不会引发错误。而对原始值 null 使用 toString()函数方法进行操作时，则会引发类型错误，同样，对于 undefined 值也会产生同样的结果，如图 4.10 所示。

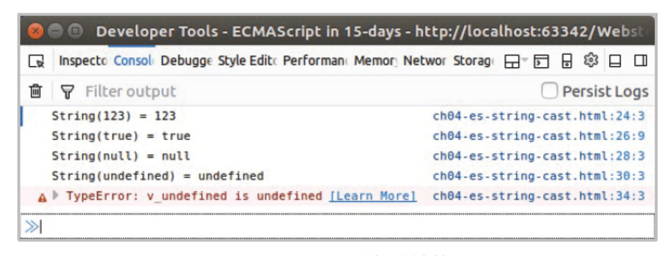


图 4.10　String()强制类型转换（2）

其实，String()强制类型转换方法是非常简单的，该方法可把任何值转换为字符串。不过，还是建议使用 toString()方法进行转换字符串的操作，除非必须使用与【代码 4-9】中第 02 行同样的代码。

众所周知，ECMAScript 是弱类型的编程语言，在很多情景下都需要对变量进行强制类型转换操作，因此本节介绍的内容是非常实用的，希望能对读者有所帮助。

4.4 本章小结

本章主要介绍了 ECMAScript 语法规范中强制类型转换方面的知识，包括 Number()、Boolean()和 String()这几个方法的使用，并通过一些具体实例进行了讲解。希望读者通过对本章内容的学习，能够打好 ECMAScript 脚本语言开发的基础。

第 5 章

解　构

本章将介绍 ECMAScript 解构方面的相关知识，特别是 ECMAScript 6 语法规范中关于变量解构赋值的内容。

5.1　ECMAScript 变量赋值机制

ECMAScript 变量赋值机制针对原始值与引用值是不一样的，这与其他程序设计语言有所区别。因此，本节将详细介绍 ECMAScript 变量赋值机制的内容。

5.1.1　变量赋值机制介绍

根据 ECMA-262 规范中的描述，ECMAScript 解释程序在处理变量赋值操作时，必须判断该值是原始类型还是引用类型。ECMAScript 解释程序在处理原始类型和引用类型的变量赋值机制上采用了不同的方式。

ECMAScript 的原始类型包括 Undefined、Null、Boolean、Number 和 String 五大类型。因此，ECMAScript 解释程序在变量赋值时，会先判断该值是否为五大类型的原始类型。而 ECMA-262 规范对于引用类型的定义比较抽象，其实引用类型就是一个对象，类似于 Java 语言中类（class）的概念。如果 ECMAScript 解释程序判断出该值不是原始类型，那就是引用类型。

这里需要特别说明一下，在许多编程语言中，字符串都是被当作引用类型来处理的。这是因为字符串的长度大小是可变的，不适于作为原始类型来处理。不过，ECMAScript 语法改变了这一点，字符串在 ECMAScript 中是作为原始类型来处理的。这样，ECMAScript 字符串的处理速度会更快。

5.1.2 变量赋值机制相关原理

由于原始类型的值所占据的空间是固定的，因此可将它存储在占用较小内存区域的“栈”中，这种存储机制便于 ECMAScript 解释程序迅速查寻变量的值。如果一个值是引用类型，那么其存储空间将从“堆”中分配。

“栈”与“堆”是计算机操作系统中两个十分重要的概念。从数据结构上理解，“栈”是一种“先进后出、后进先出”的存储结构，“堆”是一种树形存储结构。从计算机操作系统原理上理解，“栈”一般位于一级缓存中，“堆”一般位于二级缓存中，一级缓存的存取速度快于二级缓存。

“栈”与“堆”的结构关系到变量的存储机制。由于 ECMAScript 引用值的大小会改变，因此不能将其存储在“栈”中，否则会降低变量查寻访问的速度。而指向引用值的指针（pointer）是存储在“栈”中的，该值是该引用值对象存储在“堆”中的地址，因为地址的大小是固定的，所以将其存储在“栈”中没有任何问题。

5.1.3 关于变量的解构赋值

在 ECMAScript 6 语法规范中新增了一种被称为“解构（Destructuring）赋值”的方式。具体来说，就是 ECMAScript 6 语法允许按照一定模式，通过从数组或对象中提取值对变量进行赋值，因此这种方式也被称为解构（Destructuring）。解构或解构赋值涉及 ECMAScript 的各种变量类型（数组、字符串、对象等），且不同变量类型的解构方式均略有不同。

下面先看一个 ECMAScript 解构赋值的简单代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring.html 文件）。

【代码 5-1】

```
01  <script type="text/javascript">
02      "use strict";
03      /*
04       * 数组赋值
05       */
06      var arr = [1, 2, 3];   // TODO: define array
07      var a = arr[0];
08      var b = arr[1];
09      var c = arr[2];
10      console.log("print - a, b, c : ");
11      console.log(a, b, c);
12     /*
13      * 解构赋值
14      */
15      [c, b, a] = arr;   // TODO: array destructuring
16     console.log("print after destructuring - a, b, c : ");
17     console.log(a, b, c);
18  </script>
```

关于【代码 5-1】的分析如下：

第 06 行代码定义了一个简单数组 arr，同时进行了初始化赋值操作。

第 07～09 行代码中依次将数组 arr 的每个数组项按顺序赋值给了三个变量 a、b 和 c。

第 15 行代码通过数组解构赋值的方式对以上 3 个变量（a、b 和 c）重新进行了赋值操作。

运行页面，调试信息如图 5.1 所示。经过解构赋值操作后的三个变量 a、b、c 值的顺序发生了变化。

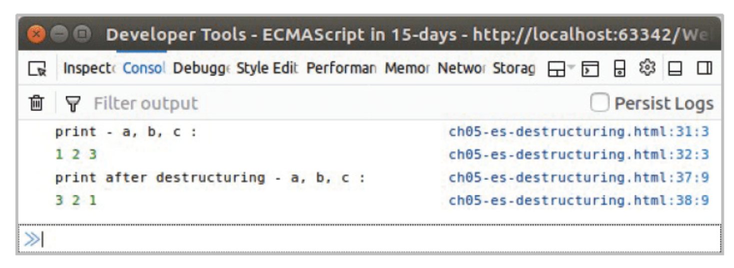


图 5.1　解构赋值的基本方式

如果仅仅从代码上看，解构赋值方式要比重新对三个变量 a、b、c 一步一步重新赋值的传统方式简单多了，这一点正是解构赋值的强大之处。

5.2　ECMAScript 数组解构赋值

本节将介绍 ECMAScript 语法规范中关于数组的解构赋值，包括数组解构赋值的基本方式和特殊用法。

5.2.1　数组解构赋值的基本方式

ECMAScript 语法规范中的数组解构赋值基本是按照等号左边与等号右边的匹配来进行的。其基本语法结构如下：

```
[var1, var2, var3, … , varN] = arr    // TODO: varN 表示一个变量，arr 表示一个数组
```

下面看一个数组解构赋值基本方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr.html 文件）。

【代码 5-2】

```
01  <script type="text/javascript">
02      "use strict";
03      /*
04       * 数组解构赋值
05       */
06      var [a, b, c] = [1, 2, 3];    // TODO: array destructuring
07      console.log("Array destructuring - a, b, c : ");
08      console.log("a = " + a);
09      console.log("b = " + b);
10      console.log("c = " + c);
11  </script>
```

关于【代码 5-2】的分析如下：

第 06 行代码通过数组解构赋值的方式，在定义了一组变量 a、b、c 的同时进行了初始化赋值操作。

运行页面，调试信息如图 5.2 所示。在通过数组解构赋值的方式对一组变量 a、b、c 进行初始化操作后，变量 a、b、c 依次被初始化赋值为数组[1, 2, 3]的取值。这与传统变量赋值方式相比，数组解构赋值方式仅仅通过【代码 5-2】中第 06 行这一行代码就实现了对三个变量的初始化赋值操作。

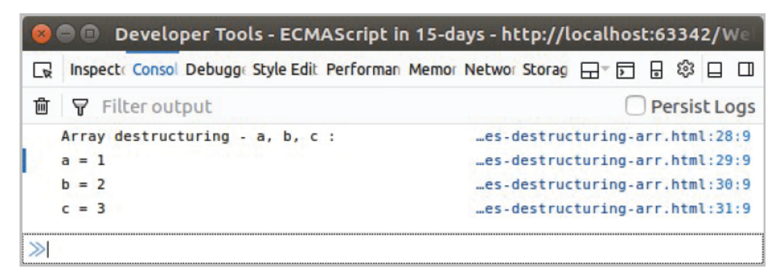


图 5.2　数组解构赋值的基本方式

5.2.2　数组解构赋值的嵌套方式

ECMAScript 语法规范中的数组解构赋值，除了前面介绍的基本方式外，还支持任意深度的嵌套方式。

下面看一个数组解构赋值嵌套方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr-nest.html 文件）。

【代码 5-3】

```
01  <script type="text/javascript">
02      "use strict";
03      /*
04       * 数组解构赋值的嵌套方式
05       */
06      var [a, [[b], c]] = [1, [[2], 3]];// TODO: array destructuring
07      console.log("Array destructuring : ");
08      console.log("[a, [[b], c] = [1, [[2], 3]]");
09      console.log("a = " + a);
10      console.log("b = " + b);
11      console.log("c = " + c);
12  </script>
```

关于【代码 5-3】的分析如下：

第 06 行代码通过数组解构赋值的方式，在定义了一组变量 a、b、c 的同时进行了初始化赋值操作。注意，这里的变量 b 和 c 是通过嵌套方式解构赋值的。

运行页面，调试信息如图 5.3 所示。数组解构赋值的嵌套方式支持任意深度的嵌套，只要保证正确的匹配方式就可以操作。

```
Array destructuring :
[a, [[b], c] = [1, [[2], 3]]
a = 1
b = 2
c = 3
```

图 5.3　数组解构赋值的嵌套方式

5.2.3　含有空位的数组解构赋值

ECMAScript 语法规范中的数组解构赋值，对于等号左边含有空位的变量，也可以进行相应的匹配操作。

下面看一个含有空位的数组解构赋值方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr-space.html 文件）。

【代码 5-4】

```
01  <script type="text/javascript">
02       "use strict";
03      /*
04       * 数组解构赋值
05       */
06      let [a] = [1, 2, 3];     // TODO: array destructuring
07      console.log("Array destructuring - a : ");
08      console.log("a = " + a);
09      let [b,] = [1, 2, 3];    // TODO: array destructuring
10      console.log("Array destructuring - b : ");
11      console.log("b = " + b);
12      let [i, j, ] = [1, 2, 3];    // TODO: array destructuring
13      console.log("Array destructuring - i, j : ");
14      console.log("i = " + i);
15      console.log("j = " + j);
16      let [m, , n] = [1, 2, 3];    // TODO: array destructuring
17      console.log("Array destructuring - m, n : ");
18      console.log("m = " + m);
19      console.log("n = " + n);
20      let [ , , t] = [1, 2, 3];    // TODO: array destructuring
21      console.log("Array destructuring - t : ");
22      console.log("t = " + t);
23  </script>
```

关于【代码 5-4】的分析如下：

第 06～08 行代码定义了第一组通过数组解构方式的赋值操作。需要注意的是，仅仅定义了一个变量 a，等号右侧是一个完整的数组[1, 2, 3]。

第 09～11 行代码定义了第二组通过数组解构方式的赋值操作。需要注意的是，在定义的变量 b 后面增加了一个逗号分隔符。

第 12～15 行代码定义了第三组通过数组解构方式的赋值操作。需要注意的是，定义了两个变

量（i, j,）。

第 16～19 行代码定义了第四组通过数组解构方式的赋值操作，这里同样定义了两个变量（m, ,n）。需要注意的是，变量 n 前面增加了一个逗号分隔符。

第 20～22 行代码定义了第五组通过数组解构方式的赋值操作，这里仅仅定义了一个变量(, ,t)。需要注意的是，变量 t 前面有两个逗号分隔符。

运行页面，调试信息如图 5.4 所示。

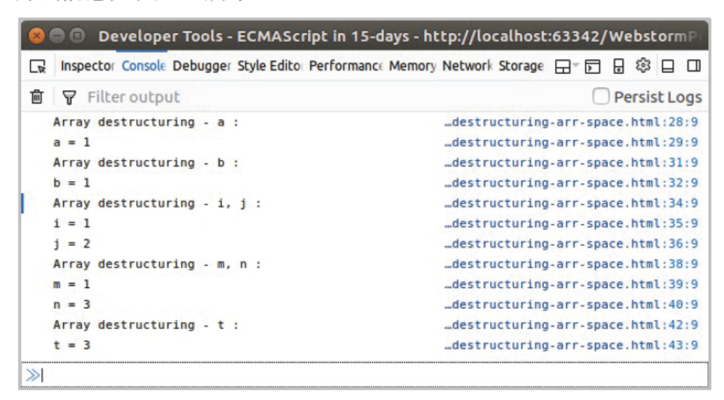


图 5.4　含有空位的数组解构赋值方式

如图 5.4 所示，尽管【代码 5-4】中变量定义的格式不完全，但通过数组解构赋值的方式均获取了各自的值。

变量 a 和变量 b 均获取了数组的第一项值 1，说明变量 b 后面增加的逗号分隔符是正确的格式。

变量 i 和变量 j 依次获取了数组的第一项和第二项值 1 和 2，说明这两个变量（i, j,）是按顺序取值的。

变量 m 和变量 n 分别获取了数组的第一项和第三项值 1 和 3，说明变量 n 前面增加的逗号分隔符起到了定义空位的作用。

同样，变量 t 获取了数组的第三项值 3，说明变量 t 前面增加的两个逗号分隔符起到了定义两个空位的作用。

【代码 5-4】演示了在数组解构赋值方式中，只要保证正确的空位匹配方式，变量就可以获取正确的值。

5.2.4　使用省略号的数组解构赋值

对于 ECMAScript 语法规范中的数组解构赋值，还可以使用省略号的方式进行相应的匹配操作。

下面看一个使用省略号数组解构赋值方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr-dots.html 文件）。

【代码 5-5】

```
01  <script type="text/javascript">
02        "use strict";
03       /*
04        * 数组解构赋值
```

```
05          */
06          let [a, ...bb] = [1, 2, 3, 4, 5];// TODO: array destructuring
07          console.log("Array destructuring - a, ...bb : ");
08          console.log("a = " + a);
09          console.log("...bb = " + bb);
10          let [i, j, ...kkk] = [1, 2, 3, 4, 5]; // TODO: array destructuring
11          console.log("Array destructuring - a, b, ...ccc : ");
12          console.log("i = " + i);
13          console.log("j = " + j);
14          console.log("...kkk = " + kkk);
15      </script>
```

关于【代码 5-5】的分析如下：

第 06～09 行代码定义了第一组通过数组解构方式的赋值操作，定义了两个变量 a 和 bb。这里需要注意的是，变量 bb 前面增加了省略号修饰符，等号右侧是一个完整的数组[1, 2, 3, 4, 5]。

第 10～14 行代码定义了第二组通过数组解构方式的赋值操作，定义了三个变量 i、j 和 kkk，同样是变量 kkk 前面增加了省略号修饰符，等号右侧是一个完整的数组[1, 2, 3, 4, 5]。

运行页面，调试信息如图 5.5 所示。

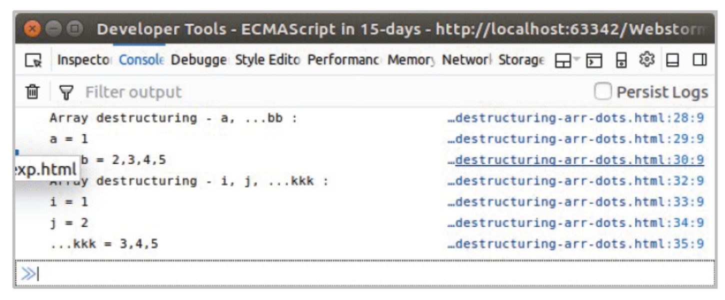


图 5.5　使用省略号的数组解构赋值方式

如图 5.5 所示，变量 a 获取了数组的第一项值 1，使用了省略号修饰符的变量 bb 获取了数组的其余值（2, 3, 4, 5）。

变量 i 获取了数组的第一项值 1，变量 j 获取了数组的第二项值 2，使用了省略号修饰符的变量 kkk 获取了数组的其余值（3, 4, 5）。

通过变量 bb 和变量 kkk 的取值可以清楚地看到，使用了省略号修饰符的变量获取了数组的剩余项的值。

在 ECMAScript 语法规范中，使用省略号修饰符的数组解构赋值是有格式要去的，也就是说带有省略号修饰符的变量必须放在最后，否则是无效的解构方式。

下面看一个无效使用省略号数组解构赋值方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr-dots-err.html 文件）。

【代码 5-6】

```
01      <script type="text/javascript">
02           "use strict";
03          /*
04           * 数组解构赋值
05           */
06          let [m, ...e, n] = [1, 2, 3, 4, 5];   // TODO: array destructuring
```

```
07        console.log("Array destructuring - m, ...e, n : ");
08        console.log("m = " + m);
09        console.log("...e = " + e);
10        console.log("n = " + n);
11    </script>
```

关于【代码 5-6】的分析如下：

第 06～10 行代码定义了一组通过数组解构方式的赋值操作，定义了 3 个变量：m、e 和 n。这里需要注意的是，变量 e 前面增加了省略号修饰符，但变量 e 并不是最后一个变量。

运行页面，调试信息如图 5.6 所示。如图 5.6 中箭头所指，浏览器控制台中提示了错误信息（不是一个结尾变量），表明使用了省略号修饰符的变量 e 无法正确获取数组项的值。

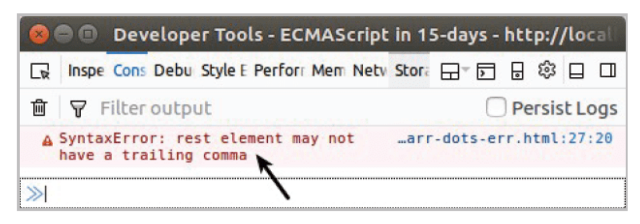


图 5.6　无效使用省略号的数组解构赋值方式

5.2.5　未定义的数组解构赋值

对于 ECMAScript 语法规范中的数组解构赋值，如果变量未获得有效的匹配操作，变量就会被赋予未定义（undefined）原始值。

下面看一个未定义的数组解构赋值方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr-undefined.html 文件）。

【代码 5-7】

```
01    <script type="text/javascript">
02        "use strict";
03        /*
04         * 数组解构赋值
05         */
06        var [a] = [];  // TODO: array destructuring
07        var [b, c] = [1];  // TODO: array destructuring
08        console.log("Array destructuring - a, b, c : ");
09        console.log("a = " + a);
10        console.log("b = " + b);
11        console.log("c = " + c);
12    </script>
```

关于【代码 5-7】的分析如下：

第 06 行代码通过数组解构方式定义了一个变量 a，注意在等号右侧是一个空数组[]。

第 07 行代码通过数组解构方式定义了两个变量 b 和 c，注意在等号右侧虽然不是一个空数组，但仅含有一个数组项[1]。

运行页面，调试信息如图 5.7 所示。变量 a 最终被定义为未定义（undefined）原始值。同样，变量 c 最终也被定义为未定义（undefined）原始值，仅仅是变量 b 最终获取了有效数值 1。

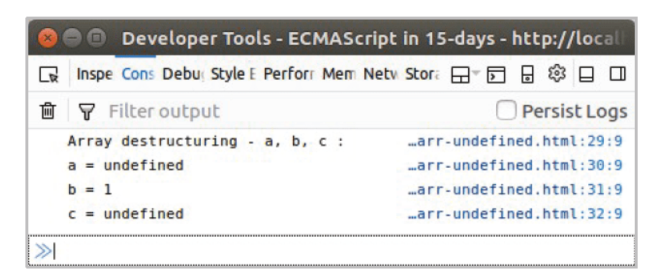


图 5.7　未定义的数组解构赋值方式

通过以上的输出结果表明，在数组解构赋值方式中，当等号左侧的变量数目多于等号右侧的数组项时，在匹配操作完成后，多余的变量是无法获取有效数值的，这些变量将会被赋予未定义（undefined）原始值。从严格意义上来说，变量被赋予未定义（undefined）原始值并不算是一个语法错误。

5.2.6　无效的数组解构赋值

对于 ECMAScript 语法规范中的数组解构赋值，等号左侧的变量与右侧的数组是要遵循一定的语法规则的，否则将会出现无效的数组解构赋值问题。

下面看一个无效的数组解构赋值方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr-invalid-num.html 文件）。

【代码 5-8】

```
01  <script type="text/javascript">
02       "use strict";
03      /*
04       * 无效的数组解构赋值
05       */
06      let [a] = 1;
07      console.log("Array destructuring - a : ");
08      console.log("a = " + a);
09  </script>
```

关于【代码 5-8】的分析如下：

第 06 行代码通过数组解构方式定义了一个变量 a，注意在等号右侧是一个数值 1，而不是一个数组。

运行页面，调试信息如图 5.8 所示。浏览器控制台中给出了错误提示信息（数值 1 并不是一个迭代类型数据），这就是数组解构赋值的语法要求。

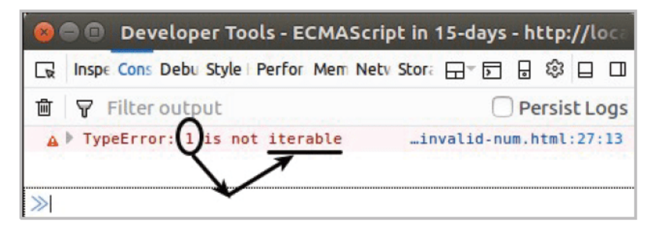


图 5.8　无效的数组解构赋值方式（1）

同样，再看下面这个无效的数组解构赋值方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr-invalid-boolean.html 文件）。

【代码 5-9】

```
01  <script type="text/javascript">
02       "use strict";
03      /*
04       * 无效的数组解构赋值
05       */
06      let [b] = true;
07      console.log("Array destructuring - b : ");
08      console.log("b = " + b);
09  </script>
```

关于【代码 5-9】的分析如下：

第 06 行代码通过数组解构方式定义了一个变量 b，注意在等号右侧是一个布尔值 true，而不是一个迭代类型。

运行页面，调试信息如图 5.9 所示。浏览器控制台中给出了同样的错误提示信息（布尔值 true 不是一个迭代类型数据）。

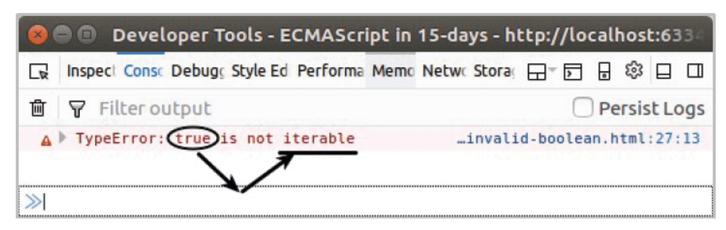


图 5.9　无效的数组解构赋值方式（2）

现在问题就很清楚了，有效的数组解构赋值方式必须使用可迭代的类型，否则就会出现语法错误。因此，对于那些特殊的原始值，是无法实现数组解构赋值的。

下面继续看一个无效的数组解构赋值方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr-invalid-spec.html 文件）。

【代码 5-10】

```
01  <script type="text/javascript">
02       "use strict";
03      /*
04       * 无效的数组解构赋值
05       */
06      let [nan] = NaN;
07      console.log("Array destructuring - nan : ");
08      console.log("nan = " + nan);
09      let [u] = undefined;
10      console.log("Array destructuring - u : ");
11      console.log("u = " + u);
12      let [nul] = null;
13      console.log("Array destructuring - nul : ");
14      console.log("nul = " + nul);
15  </script>
```

关于【代码 5-10】的分析如下：

这段代码对几个特殊的原始值 NaN、undefined 和 null 进行了数组解构赋值的测试，效果与前面两个代码示例基本相同。

运行页面，调试信息如图 5.10 所示。浏览器控制台中给出了同样的错误提示信息（特殊值 NaN 不是一个迭代类型数据）。

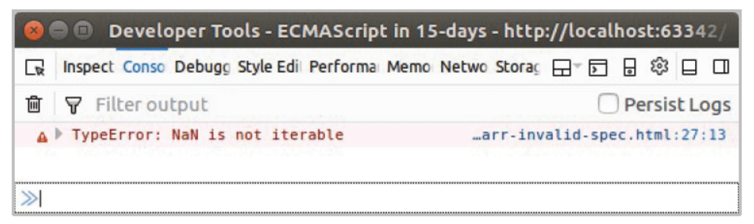


图 5.10　无效的数组解构赋值方式（3）

5.2.7　使用默认值的数组解构赋值

对于 ECMAScript 语法规范中的数组解构赋值，还支持使用默认值的方式进行相应的匹配操作。对于使用默认值的数组解构赋值，等号右侧的数组格式必须使用严格的特殊原始值 undefined。

下面看一个使用默认值的数组解构赋值方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr-default.html 文件）。

【代码 5-11】

```
01  <script type="text/javascript">
02       "use strict";
03      /*
04       * 使用默认值的数组解构赋值
05       */
06      let [a] = [];
07      console.log("Array destructuring - a : ");
08      console.log("a = " + a);
09      let [b] = [undefined];
10      console.log("Array destructuring - b : ");
11      console.log("b = " + b);
12      let [c = 1] = [];
13      console.log("Array destructuring - c : ");
14      console.log("c = " + c);
15      let [d = 1] = [undefined];
16      console.log("Array destructuring - d : ");
17      console.log("d = " + d);
18      let [e = 1] = [2];
19      console.log("Array destructuring - e : ");
20      console.log("e = " + e);
21  </script>
```

关于【代码 5-11】的分析如下：

第 06 行代码通过数组解构方式定义了第一个变量 a，等号右侧是一个空的数组[]。

第 09 行代码通过数组解构方式定义了第二个变量 b，等号右侧是一个数组[undefined]。

第 12 行代码通过数组解构方式定义了第三个变量 c，这里与第 06 行代码的方式类似，唯一区别是等号左侧的变量 c 赋予了默认值 1。

第 15 行代码通过数组解构方式定义了第四个变量 d，这里与第 09 行代码的方式类似，唯一区别是等号左侧的变量 d 赋予了默认值 1。

第 18 行代码通过数组解构方式定义了第五个变量 e，等号左侧的变量 e 赋予了默认值 1，等号右侧是一个数组[2]。

运行页面，调试信息如图 5.11 所示。

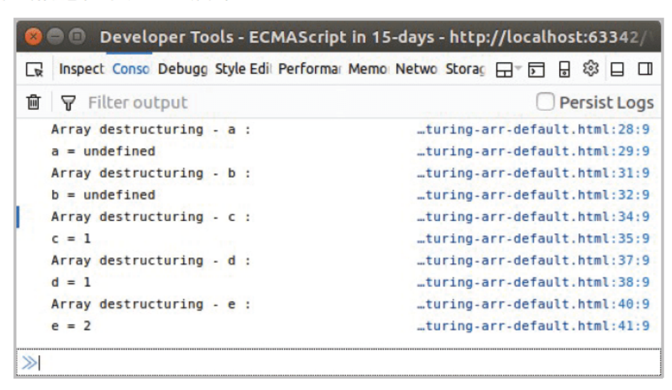


图 5.11　使用默认值的数组解构赋值方式

如图 5.11 所示，变量 a 和变量 b 均获取了原始值 undefined，说明即使在数组解构方式下定义一个空的数组，变量仍会获取原始值 undefined。

变量 c 获取了数值 1，说明在数组解构方式下定义一个空的数组，变量会获取定义的默认值[c = 1]。

变量 d 在等号左侧定义了默认值 1，最终也是获取了数值 1，但在等号右侧的数组内定义了原始值 undefined，这一点很关键（请与下面变量 e 的结果进行对比）。

变量 e 与变量 d 类似，在等号左侧定义了默认值 1，但在等号右侧的数组内定义了数值 2，最终结果也是获取了数值 2。

通过变量 d 和变量 e 的结果对比，可以看到在使用默认值的数组解构赋值方式下，当等号右侧的数组定义为原始值 undefined 或为空时，变量才会获取等号左侧定义的默认值。

5.2.8　默认值为变量的数组解构赋值

在 5.2.7 小节中介绍了使用默认值的数组解构赋值方法，其实该默认值还支持使用其他变量的方式进行操作，前提是其他变量必须是已经定义过的。

下面看一个默认值为变量的数组解构赋值方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr-default-var.html 文件）。

【代码 5-12】

```
01    <script type="text/javascript">
02        "use strict";
03        /*
```

```
04         * 默认值为变量的数组解构赋值
05         */
06        let [a, b = a] = [1];
07        console.log("Array destructuring - a, b = a: ");
08        console.log("a = " + a);
09        console.log("b = " + b);
10        let [c = 1, d = c] = [];
11        console.log("Array destructuring - c = 1, d = c: ");
12        console.log("c = " + c);
13        console.log("d = " + d);
14        let [e = 1, f = e] = [2];
15        console.log("Array destructuring - e = 1, f = e: ");
16        console.log("e = " + e);
17        console.log("f = " + f);
18        let [g = 1, h = g] = [1, 2];
19        console.log("Array destructuring - g = 1, h = g: ");
20        console.log("g = " + g);
21        console.log("h = " + h);
22        let [i = j, j = 1] = [];
23        console.log("Array destructuring - i = j, j = 1: ");
24        console.log("i = " + i);
25        console.log("j = " + j);
26    </script>
```

关于【代码 5-12】的分析如下：

第 06 行代码通过数组解构方式定义了第一组变量 a 和 b，变量 b 使用了默认值赋值方式，且赋予的是变量 a 的值。

第 10 行代码通过数组解构方式定义了第二组变量 c 和 d，变量 c 使用了默认值赋值方式 c=1，变量 d 使用了默认值赋值方式，且赋予的是变量 c 的值。

第 14 行代码通过数组解构方式定义了第三组变量 e 和 f，其与第 10 行代码类似，唯一区别是等号右侧定义的数组项数值 2。

第 18 行代码通过数组解构方式定义了第四组变量 g 和 h，变量 g 使用了默认值赋值方式 g=1，变量 h 使用了默认值赋值方式，且赋予的是变量 g 的值，等号右侧定义了一个数组[1, 2]。

第 22 行代码通过数组解构方式定义了第五组变量 i 和 j，变量 i 使用了默认值赋值方式 i=j，变量 j 使用了默认值赋值方式 j=1。

运行页面，调试信息如图 5.12 所示。

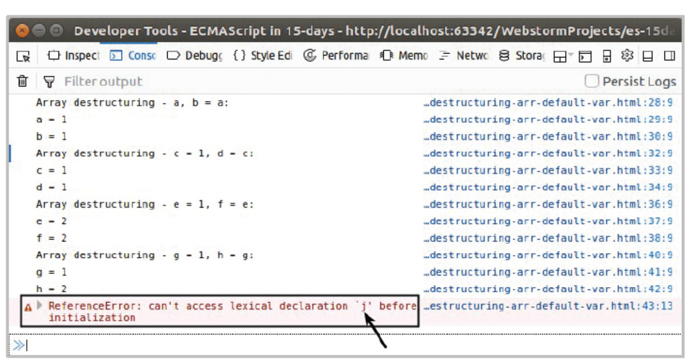


图 5.12 默认值为变量的数组解构赋值方式

如图 5.12 所示，变量 a 和变量 b 均获取了数值 1，说明变量 b 通过默认值为变量的方式取得了变量 a 的值。

变量 c 和变量 d 均获取了数值 1，说明变量 c 先通过默认值的方式取得了数值 1，然后变量 d 通过默认值为变量的方式取得了变量 c 的值。

变量 e 和变量 f 均获取了数值 2，说明变量 f 通过默认值为变量的方式取得了变量 e 的值。

变量 g 和变量 h 分别获取了数值 1 和 2，说明变量 g 先通过默认值的方式取得了数值 1，然后变量 h 尝试通过默认值为变量的方式取得变量 g 的值，但操作未成功，因为等号右侧的数组中定义了数字项（数值为 2），所以变量 h 获取了该数值 2。

再看图 5.12 中箭头所指，浏览器控制台给出错误信息表示变量 j 在使用时还未初始化，因此在使用默认值为变量的数组解构赋值方式中，该变量必须已经进行了初始化操作才会成功。

5.2.9 默认值为表达式的数组解构赋值

使用默认值的数组解构赋值方法时，该默认值还支持使用表达式的方式进行操作，其实原则上与默认值为变量的方式类似。

下面看一个默认值为表达式的数组解构赋值方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-arr-default-exp.html 文件）。

【代码 5-13】

```
01  <script type="text/javascript">
02       "use strict";
03      /*
04       * 默认值为表达式的数组解构赋值
05       */
06      function fn_arr_destructuring() {
07           return "fn_arr_destructuring()";
08      }
09      let [f = fn_arr_destructuring()] = ["f"];
10      console.log("Array destructuring - f : ");
11      console.log("f = " + f);
12      let [fn = fn_arr_destructuring()] = [];
13      console.log("Array destructuring - fn : ");
14      console.log("fn = " + fn);
15  </script>
```

关于【代码 5-13】的分析如下：

第 06～08 行代码定义了一个函数方法 fn_arr_destructuring()，该函数很简单，仅仅返回了一行字符串信息。

第 09 行代码通过默认值为表达式的数组解构方式定义了第一个变量 f，其表达式为第 06～08 行代码定义的函数方法 fn_arr_destructuring()，等号右侧的数组定义了一个字符串数组项 f。

第 12 行代码再次通过默认值为表达式的数组解构方式定义了第二个变量 fn，其表达式仍为函数方法 fn_arr_destructuring()，注意与第 09 行代码不同的是，等号右侧的数组定义了一个空数组。

运行页面，调试信息如图 5.13 所示。

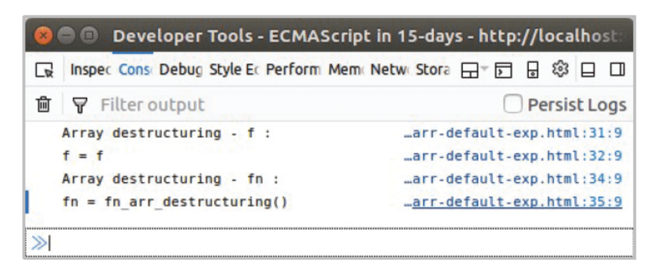


图 5.13　默认值为表达式的数组解构赋值方式

如图 5.13 所示，变量 f 获取了字符串“f”，说明变量 f 尝试通过默认值为表达式的数组解构赋值方式未成功。

而变量 fn 获取了函数方法 fn_arr_destructuring()返回的字符串“fn_arr_destructuring()”，说明变量 fn 尝试通过默认值为表达式的数组解构赋值方式取得了成功。

通过变量 f 和变量 fn 的结果对比，可以看到在使用默认值为表达式的数组解构赋值方式中，变量只有在未取得等号右侧数组中的具体值时，才能够取得该表达式的值（或表达式才会被操作执行）。

5.3　ECMAScript 对象解构赋值

本节将介绍 ECMAScript 语法规范中关于对象的解构赋值，包括对象解构赋值的基本方式和特殊用法。

5.3.1　对象解构赋值的基本方式

ECMAScript 语法规范中的对象解构赋值同样是按照等号左边与等号右边的匹配来进行的。对象解构赋值与数组解构赋值的区别是，数组是按照次序进行匹配的，而对象是按照属性名匹配的（不一定按照先后次序）。

关于对象解构赋值的基本语法结构如下：

```
// TODO: 对象解构赋值是按照等号左右两边的属性名进行匹配的
{属性名 1:变量 1, 属性名 2:变量 2, ..., 属性名 n:变量 n} =
{属性名 1:属性值 1, 属性名 2:属性值 2, ..., 属性名 n:属性值 n}
```

下面看一个对象解构赋值基本方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-obj.html 文件）。

【代码 5-14】

```
01  <script type="text/javascript">
02       "use strict";
03      /*
04       * 对象解构赋值
05       */
06      var {oA, oB} = {oA : "AAA", oB : "BBB"};   // TODO: object destructuring
07      console.log("object destructuring - oA, oB : ");
```

```
08      console.log("oA = " + oA);
09      console.log("oB = " + oB);
10  </script>
```

关于【代码 5-14】的分析如下：

第 06 行代码为对象解构赋值的方式，在等号左侧以对象形式定义了两个变量 oA 和 oB，在等号右侧定义了一个对象，同时对变量 oA 和 oB 进行了初始化赋值操作。

运行页面，调试信息如图 5.14 所示。

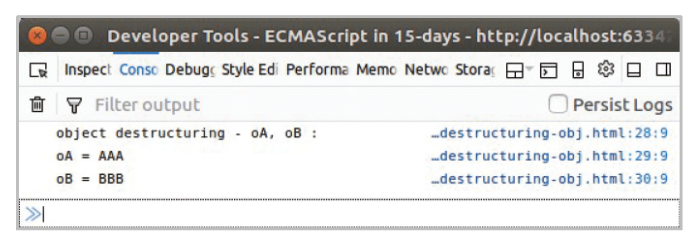


图 5.14　对象解构赋值的基本方式

如图 5.14 所示，在通过对象解构赋值的方式对两个变量 oA 和 oB 进行初始化操作后，变量 oA 和 oB 依次获取了等号右侧对象中的值"AAA"和"BBB"。

这与传统变量赋值方式相比，对象解构赋值方式仅仅通过【代码 5-14】中第 06 行这一行代码就实现了对一组变量的初始化赋值操作。

通过前面对数组解构赋值方式的学习，读者一定会想，如果将【代码 5-14】中的对象解构赋值改成通过数组解构方式赋值来实现，效果不是一样的吗，两者有什么区别呢？其实这两种解构方式还是有明显区别的，请继续阅读下面的内容。

5.3.2　不按次序的对象解构赋值

ECMAScript 语法规范中描述的对象解构赋值方式，特别提到了是按照属性名来进行匹配的，也就是不一定要按照先后次序进行匹配。

下面看一个不按次序进行对象解构赋值方式的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-obj-disorder.html 文件）。

【代码 5-15】

```
01  <script type="text/javascript">
02       "use strict";
03      /*
04       * 不按次序的对象解构赋值
05       */
06      var {oA, oB} = {oB : "AAA", oA : "BBB"};   // TODO: object destructuring
07      console.log("object destructuring - oA, oB : ");
08      console.log("oA = " + oA);
09      console.log("oB = " + oB);
10      var {oD, oC} = {oC : "CCC", oD : "DDD"};   // TODO: object destructuring
11      console.log("object destructuring - oC, oD : ");
12      console.log("oC = " + oC);
13      console.log("oD = " + oD);
```

```
14  </script>
```

关于【代码 5-15】的分析如下：

第 06 行代码为对象解构赋值的方式，在等号左侧通过对象方式定义了两个变量 oA 和 oB，在等号右侧定义了一个对象（注意对象中的属性名 oA 和 oB 的顺序与变量 oA 和 oB 的顺序是不一致的），同时对变量 oA 和 oB 进行了初始化赋值操作。

第 10 行代码与第 06 行代码类似，在等号左侧通过对象方式定义的两个变量 oD 和 oC 的顺序，与等号右侧定义的对象中属性名的顺序是不一致的。

运行页面，调试信息如图 5.15 所示。在通过对象解构赋值的方式对两个变量 oA 和 oB 进行初始化操作后，变量 oA 和 oB 依次获取了等号右侧对象中的值"BBB"和"AAA"，这就说明在对象解构赋值方式中是按照变量名和属性名来匹配的，与变量和属性定义的次序没有关系。同理，另外两个变量 oC 和 oD 获取了属性值"CCC"和"DDD"也就很好理解了。

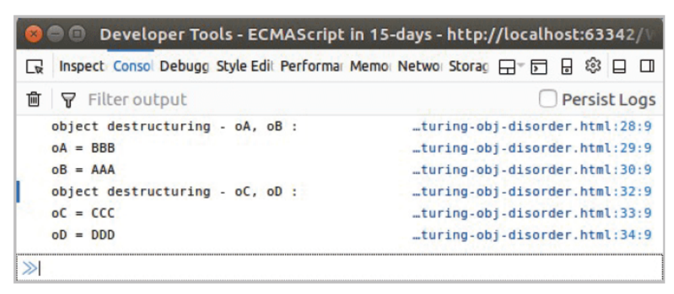


图 5.15　不按次序的对象解构赋值方式

5.3.3　对象解构赋值方式的扩展

在前面介绍了关于对象解构赋值方式的语法形式，与【代码 5-14】和【代码 5-15】中的对象解构赋值方式似乎有所出入。其实，【代码 5-14】和【代码 5-15】是对象解构赋值的简写方式，下面要介绍的语法形式是对象解构赋值方式的扩展，也可以理解为完整方式。

下面看一个对象解构赋值方式扩展的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-obj-extend.html 文件）。

【代码 5-16】

```
01  <script type="text/javascript">
02      "use strict";
03      /*
04       * 对象解构赋值方式的扩展
05       */
06     var {oA : oA, oB : oB} = {oA : "AAA", oB : "BBB"};// TODO: object destructuring
07      console.log("object destructuring - oA, oB : ");
08      console.log("oA = " + oA);
09      console.log("oB = " + oB);
10     var {oC : oc, oD : od} = {oC : "CCC", oD : "DDD"};// TODO: object destructuring
11      console.log("object destructuring - oc, od : ");
12      console.log("oc = " + oc);
13      console.log("od = " + od);
14      console.log("object destructuring - oC, oD : ");
```

```
15        console.log("oC = " + oC);
16        console.log("oD = " + oD);
17    </script>
```

关于【代码 5-16】的分析如下：

第 06 行代码为对象解构赋值的扩展方式，在等号左侧定义的对象包含了两个属性(oA : oA, oB : oB），且属性名与属性值是相同的；在等号右侧定义了一个对象（oA : "AAA", oB : "BBB"）。

第 10 行代码与第 06 行代码类似，不同之处在于等号左侧定义的对象所包含的两个属性（oC : oc, oD : od），属性名与属性值是不相同的。

运行页面，调试信息如图 5.16 所示。

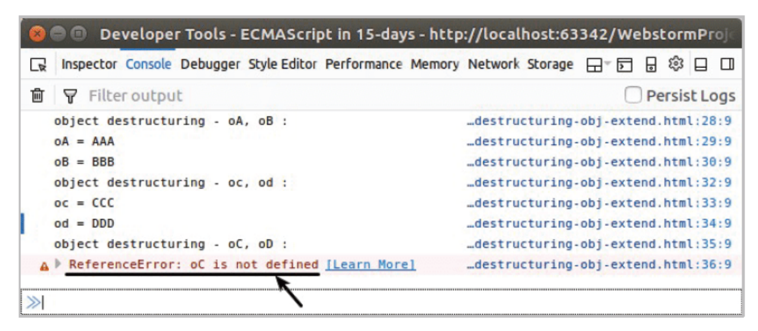


图 5.16　对象解构赋值的扩展方式

如图 5.16 所示，在通过对象解构赋值的扩展方式对两个变量 oA 和 oB 进行初始化操作后，变量 oA 和 oB 依次获取了等号右侧对象中的值"AAA"和"BBB"，这个似乎看不出有什么特别之处。

但是，在通过对象解构赋值的扩展方式将对象 oC : oc, oD : od 进行初始化操作后，结果是属性值 oc 和 od 获取了等号右侧对象中的值"CCC"和"DDD"。如图 5.16 中箭头所指，浏览器控制台中的调试信息提示属性名 oC 是“未定义”的，由此推断属性名 oD 也是“未定义”的。

通过属性名 oC 和 oD 的结果可以看到，在对象解构赋值方式中属性名是用来与对象匹配的，而属性值具体指代变量名。

5.4　ECMAScript 字符串解构赋值

本节将介绍 ECMAScript 语法规范中关于字符串的解构赋值，在对字符串进行解构赋值时，通常是先将字符串转换为对象后再处理。

下面看一个字符串解构赋值的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-string.html 文件）。

【代码 5-17】

```
01    <script type="text/javascript">
02        "use strict";
03        /*
04         * 字符串的解构赋值
05         */
```

```
06        let [a, b, c] = "abc"; // TODO: String destructuring
07      console.log("print after destructuring - a, b, c : ");
08      console.log("a = " + a);
09      console.log("b = " + b);
10      console.log("c = " + c);
11   </script>
```

关于【代码 5-17】的分析如下：

第 06 行代码就是字符串解构赋值方式的实现，在等号左侧以数组的形式定义了一组变量 a、b 和 c，在等号右侧定义了一个字符串，此时会先将该字符串转换为一个字符数组的形式。

运行页面，调试信息如图 5.17 所示。变量 a、b 和 c 在通过解构赋值操作后，分别获取了值"a" "b"和"c"。

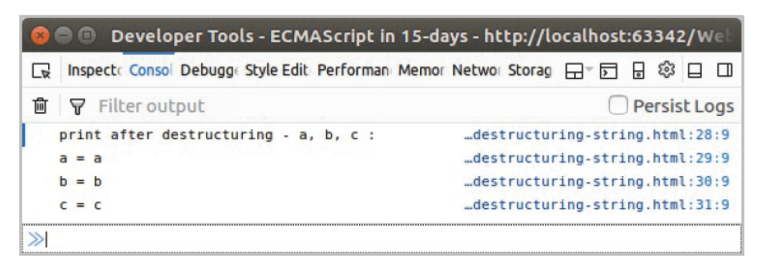


图 5.17　字符串的解构赋值方式

5.5　ECMAScript 数值解构赋值

本节将介绍 ECMAScript 语法规范中关于数值的解构赋值，在进行对象解构赋值操作时，如果等号右侧是数值，就会先将该数值转换为对象后再处理。

下面看一个字符串解构赋值的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-num.html 文件）。

【代码 5-18】

```
01   <script type="text/javascript">
02        "use strict";
03        /*
04         * 数值的解构赋值
05         */
06        let {toString : s} = 123;    // TODO: String destructuring
07      console.log("print after destructuring - s : ");
08      if(s === Number.prototype.toString)
09         console.log("s = " + s);
10   </script>
```

关于【代码 5-18】的分析如下：

第 06 行代码就是数值解构赋值方式的实现，在等号左侧以对象的方式定义了一个变量 s，在等号右侧定义了一个数值 123。

第 08 和第 09 行代码判断变量 s 是否全等于 Number 原型对象 toString，根据判断结果输出变

量 s 的内容。

运行页面，调试信息如图 5.18 所示。变量 s 全等于原型对象 Number.prototype.toString，这就说明数值 123 先转换为了 Number 对象，而变量 s 获取了 Number 对象原型的 toString 属性。

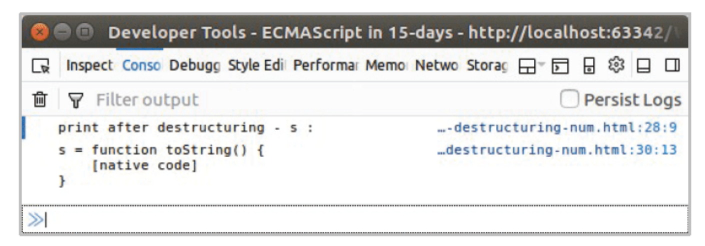


图 5.18　数值的解构赋值方式

5.6　ECMAScript 解构赋值的应用

本节将介绍 ECMAScript 解构赋值的应用，包括交换变量的值、函数多个返回值、定义函数参数等内容。

5.6.1　交换变量的值

通过 ECMAScript 数组解构赋值的方式，可以很方便地实现对变量交换值的操作，相比于传统的交换方法有效地提高了效率。

下面是一个关于通过数组解构赋值的方式实现变量交换值的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-var-exchange.html 文件）。

【代码 5-19】

```
01  <script type="text/javascript">
02      "use strict";
03      /*
04       * 传统方式交换变量的值
05       */
06      console.log("------------ 传统方式交换变量的值 ------------");
07      var a = 1;
08      var b = 2;
09      console.log("a = " + a);
10      console.log("b = " + b);
11      console.log("Exchange a & b 's data by ordinary method :");
12      var temp = a;
13      a = b;
14      b = temp;
15      console.log("a = " + a);
16      console.log("b = " + b);
17      /*
18       * 通过数组解构方式实现交换变量的值
19       */
```

```
20        console.log("---------- 数组解构方式实现交换变量的值 ----------");
21        var [x, y] = [1, 2];    // TODO: array destructuring
22        console.log("Array destructuring - x, y : ");
23        console.log("x = " + x);
24        console.log("y = " + y);
25        console.log("Exchange x & y 's data with array destructuring :");
26        [x, y] = [y, x];    // TODO: Exchange data
27        console.log("x = " + x);
28        console.log("y = " + y);
29    </script>
```

关于【代码 5-19】的分析如下：

第 06～16 行代码通过传统方式(借助临时中间变量)实现了交换变量的值，关键部分是第 12～14 行定义的代码。

第 20～28 行代码通过数组解构赋值实现了交换变量的值，关键部分是第 26 行定义的代码，这里通过数组解构赋值方式交换了变量 x 和变量 y 的值。

运行页面，调试信息如图 5.19 所示。通过数组解构赋值方式交换了变量 x=1 和变量 y=2 的值，交换后的结果为 x=2、y=1。相比第 12～14 行代码定义的传统方式，数组解构赋值方式仅仅通过第 26 行定义的这一行代码就实现了变量值的交换。

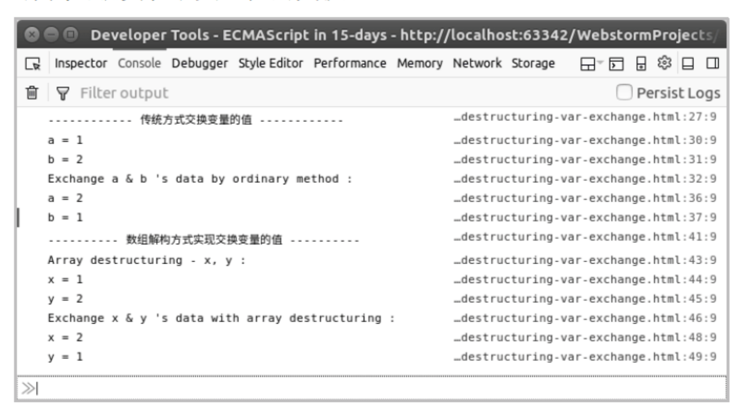


图 5.19　通过数组解构方式交换变量的值（1）

【代码 5-19】仅仅是针对两个变量实现交换值的情况，可能效果还不太明显。如果是针对多个变量（3 个或 3 个以上），那么数组解构赋值方式的优势就会愈发明显。

下面就是一个关于通过数组解构赋值的方式实现 3 个变量交换值的代码示例(详见源代码 ch05 目录中的 ch05-es-destructuring-var-tri-exchange.html 文件）。

【代码 5-20】

```
01    <script type="text/javascript">
02        "use strict";
03        /*
04         * 传统方式交换变量的值
05         */
06        console.log("------------ 传统方式交换变量的值 ------------");
07        var a = 1;
08        var b = 2;
09        var c = 3;
```

```
10        console.log("a = " + a);
11        console.log("b = " + b);
12        console.log("c = " + c);
13        console.log("Exchange a & b & c 's data by ordinary method :");
14        var temp = a;
15        a = b;
16        b = c;
17        c = temp;
18        console.log("a = " + a);
19        console.log("b = " + b);
20        console.log("c = " + c);
21        /*
22         * 通过数组解构方式实现交换变量的值
23         */
24        console.log("---------- 数组解构方式实现交换变量的值 ----------");
25        var [x, y, z] = [1, 2, 3];   // TODO: array destructuring
26        console.log("Array destructuring - x, y, z : ");
27        console.log("x = " + x);
28        console.log("y = " + y);
29        console.log("z = " + z);
30        console.log("Exchange x & y & z 's data with array destructuring :");
31        [x, y, z] = [y, z, x];        // TODO: Exchange data
32        console.log("x = " + x);
33        console.log("y = " + y);
34        console.log("z = " + z);
35    </script>
```

关于【代码 5-20】的分析如下：

第 06～20 行代码通过传统方式（借助临时中间变量）实现了交换变量的值，关键部分是第 14～17 行定义的代码。

第 24～34 行代码通过数组解构赋值实现了交换变量的值，关键部分是第 31 行定义的代码，这里通过数组解构赋值方式交换了变量 x、变量 y 和变量 z 的值。

运行页面，调试信息如图 5.20 所示。通过数组解构赋值方式交换了变量 x=1、变量 y=2 和变量 z=3 的值，交换后的结果为 x=2、y=3、z=1。相比第 14～17 行代码定义的传统方式，数组解构赋值方式仅仅通过第 31 行定义的一行代码就实现了 3 个变量值的交换。



图 5.20　通过数组解构方式交换变量的值（2）

5.6.2　函数返回多个值

通过 ECMAScript 数组解构赋值或对象解构赋值的方式，可以很方便地实现函数返回多个值的操作，有效地扩展函数的使用方法。

下面先看一个函数返回数组的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-func-return-arr.html 文件）。

【代码 5-21】

```
01  <script type="text/javascript">
02      "use strict";
03      /*
04       * 函数返回数组
05       */
06      function func_return_arr() {
07          return ["Hello", "ECMAScript", "!"];
08      }
09      var [s1, s2, s3] = func_return_arr(); // TODO: array destructuring
10      console.log("print - s1, s2, s3 : ");
11      console.log(s1, s2, s3);
12  </script>
```

关于【代码 5-21】的分析如下：

第 06～08 行代码定义了一个函数 func_return_arr()，通过第 07 行代码返回了一个数组。

第 09 行代码通过数组解构赋值方式为变量 s1、s2 和 s3 分别初始化了函数的返回值。

运行页面，调试信息如图 5.21 所示。通过数组解构赋值方式为变量 s1、s2 和 s3 初始化了函数的返回值，操作后的结果为 s1="Hello"、s2="ECMAScript"、s1="!"。

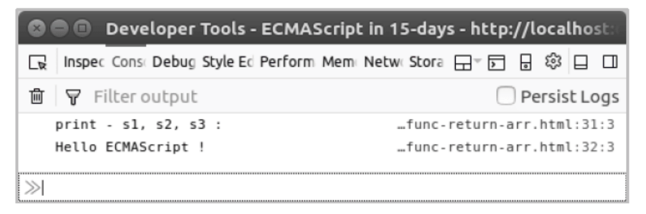


图 5.21　函数返回数组方式

下面继续看一个函数返回对象的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-func-return-obj.html 文件）。

【代码 5-22】

```
01  <script type="text/javascript">
02      "use strict";
03      /*
04       * 函数返回对象
05       */
06      function func_return_obj() {
07          return {
08              h : "Hello",
09                  e : "ECMAScript",
```

```
10                  s : "!"
11              }
12          }
13          var {h, e, s} = func_return_obj();    // TODO: object destructuring
14          console.log("print obj - h, e, s : ");
15          console.log(h, e, s);
16      </script>
```

关于【代码 5-22】的分析如下：

第 06～12 行代码定义了一个函数 func_return_obj()，通过第 07～11 行代码返回了一个对象。

第 13 行代码通过对象解构赋值方式为对象 h、e 和 s 分别初始化了函数的返回值。

运行页面，调试信息如图 5.22 所示。通过对象解构赋值方式为对象 h、e 和 s 初始化了函数的返回值，操作后的结果为 h="Hello"、e="ECMAScript"、s="!"。

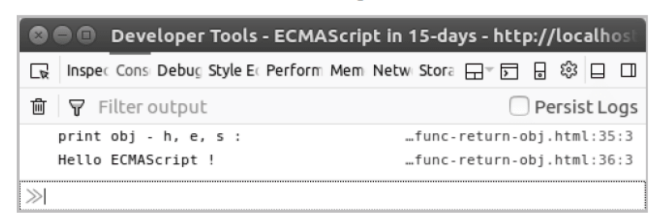


图 5.22　函数返回对象方式

5.6.3　定义函数参数

通过 ECMAScript 数组解构赋值或对象解构赋值的方式，可以很方便地实现定义函数参数的操作，有效地扩展函数的使用方法。

下面先看一个通过数组解构赋值方式，实现了定义有序函数参数的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-func-params-arr.html 文件）。

【代码 5-23】

```
01      <script type="text/javascript">
02          "use strict";
03          /*
04           * 函数有序参数
05           */
06          function func_params_arr([x, y, z]) {
07              if((x >= y) && (x >= z))
08                  return x;
09              else if(y >= z)
10                  return y;
11              else
12                  return z;
13          }
14          console.log("find max of func_params_arr([1, 2, 3] : ");
15          var max = func_params_arr([1, 2, 3]); // TODO: array destructuring
16          console.log("print - max of array = ", max);
17      </script>
```

关于【代码 5-23】的分析如下：

这段代码通过数组解构赋值方式实现了定义有序函数参数的过程，同时该函数实现了返回最大值的功能。

第 06～13 行代码定义了一个函数（func_params_arr()），其参数（[x, y, z]）定义为一个数组形式。

第 07～12 行代码通过比较排序方式，将函数的返回值定义为数组参数（[x, y, z]）中的最大值。

第 15 行代码通过数组解构赋值方式，调用了第 06～13 行代码定义的函数（func_params_arr([1, 2, 3])）。注意，这里的参数是一个有序形式。

运行页面，调试信息如图 5.23 所示。第 16 行代码成功输出了通过数组解构赋值方式定义的函数参数（[x, y, z]）中的最大值（3）。

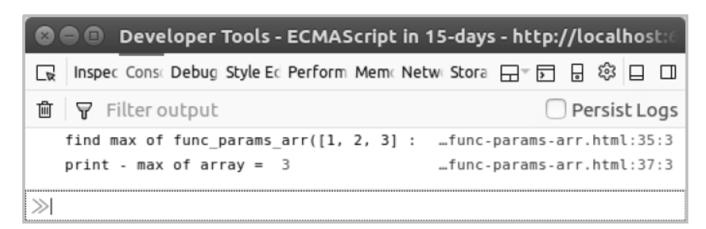


图 5.23　定义有序函数参数

下面继续看一个通过对象解构赋值方式实现定义无序函数参数的代码示例（详见源代码 ch05 目录中的 ch05-es-destructuring-func-params-obj.html 文件）。

【代码 5-24】

```
01  <script type="text/javascript">
02      "use strict";
03      /*
04       * 函数无序参数
05       */
06      function func_params_obj({x, y, z}) {
07          if((x >= y) && (x >= z))
08              return x;
09          else if(y >= z)
10              return y;
11          else
12              return z;
13      }
14     console.log("find max of func_params_obj({z:1, y:2, x:3}) : ");
15      var max = func_params_obj({z:1, y:2, x:3});    // TODO: object destructuring
16      console.log("print - max of object = ", max);
17  </script>
```

关于【代码 5-24】的分析如下：

这段代码通过对象解构赋值方式，实现了定义无序函数参数的过程，同时该函数实现了返回最大值的功能。

第 06～13 行代码定义了一个函数 func_params_obj()，其参数{x, y, z}定义为一个对象形式。

第 07～12 行代码通过比较排序方式，将函数的返回值定义为对象参数{x, y, z}中的最大值。

第 15 行代码通过对象解构赋值方式，调用了第 06～13 行代码定义的函数 func_params_obj({z:1, y:2, x:3})。注意，这里的参数是一个无序形式。

运行页面，调试信息如图 5.24 所示。第 16 行代码成功输出了通过对象解构赋值方式定义的函数参数{z:1, y:2, x:3}中的最大值 3。

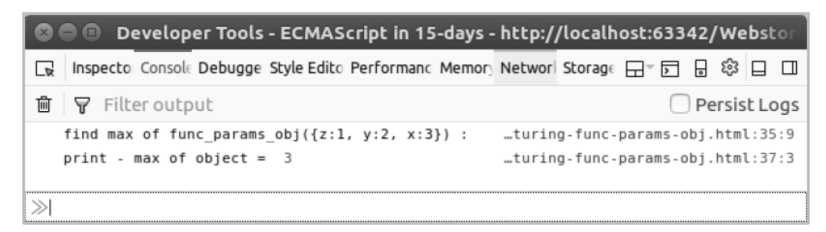


图 5.24　定义无序函数参数

5.7　本章小结

本章主要介绍了 ECMAScript 6 语法规范中关于解构的知识，包括数组解构、对象解构、字符串解构等方面的内容，并通过一些具体实例进行了讲解。希望读者通过对本章内容的学习，能够打好 ECMAScript 脚本语言开发的基础。

第 6 章

运算符与表达式

本章将介绍 ECMAScript 语法规范中关于运算符与表达式的内容。表达式是由数字、运算符、分组符号（括号）、自由变量、约束变量等元素按照一定规则组合而成的语句；运算符是用于执行表达式运算的核心元素。本章将会介绍 ECMAScript 加性运算符、乘性运算符、一元运算符、关系运算符、等性运算符、位运算符、逻辑运算符、赋值运算符、条件运算符及其表达式等方面的内容。

6.1　ECMAScript 加性运算符及表达式

ECMAScript 语法规范中将加法（+）和减法（−）运算符统一称为加性运算符，用于执行数值之间的加减算术运算。

6.1.1　概述

ECMAScript 语法规范中定义的加性运算符与表达式详见表 6.1。

表 6.1　ECMAScript 加性运算符与表达式

运算符	描述	表达式示例	运算结果
+	加	1+2	3
−	减	3−2	1

6.1.2　加法运算符及表达式

ECMAScript 语法规范中定义加法运算符使用符号“+”来表示，加法（+）运算符除了可以进行正常的数值运算外，还支持对类似“NaN”和“Infinity”特殊值的运算。此外，加法（+）运

算符还可以用于字符串的连接操作。

下面先看一个应用 ECMAScript 加法（+）运算符的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-add.html 文件）。

【代码 6-1】

```
01  <script type="text/javascript">
02      var a = 1;
03      var b = 2;
04      var sum = a + b;
05      console.log("1 + 2 = " + sum);
06      var c = NaN;
07      var d = 123;
08      var sumNaN = c + d;
09      console.log("NaN + 123 = " + sumNaN);
10      var x = Infinity;
11      var y = 321;
12      var sumInfinity = x + y;
13      console.log("Infinity + 123 = " + sumInfinity);
14      var sumX = Infinity + Infinity;
15      console.log("Infinity + Infinity = " + sumX);
16      var sumY = -Infinity + -Infinity;
17      console.log("-Infinity + -Infinity = " + sumY);
18      var sumZ = -Infinity + Infinity;
19      console.log("-Infinity + Infinity = " + sumZ);
20  </script>
```

关于【代码 6-1】的分析如下：

第 02～05 行代码定义了两个变量 a 和 b，并通过加法（+）运算符进行了算术运算。其中，在第 04 行代码中就是通过加法（+）运算符将变量 a 和变量 b 进行了算术相加运算。

第 06～09 行代码通过加法（+）运算符对特殊值（NaN）进行了算术运算。其中，第 06 行代码定义了变量 c，并初始化赋值为特殊值（NaN）；第 08 行代码尝试将变量 c 与一个具体数值进行算术相加运算，并将结果保存在变量 sumNaN 中。

第 10～13 行代码通过加法（+）运算符对特殊值（Infinity）进行了算术运算。其中，第 10 行代码定义了变量 x，并初始化赋值为特殊值（Infinity）；第 12 行代码尝试将变量 x 与一个具体数值进行算术相加运算，并将结果保存在变量 sumInfinity 中。

第 14 和第 15 行代码通过加法（+）运算符对特殊值（Infinity）和特殊值（Infinity）进行了算术运算，并将结果保存在变量 sumX 中。

第 16 和第 17 行代码通过加法（+）运算符对特殊值（-Infinity）和特殊值（-Infinity）进行了算术运算，并将结果保存在变量 sumY 中。

第 18 和第 19 行代码通过加法（+）运算符对特殊值（-Infinity）和特殊值（Infinity）进行了算术运算，并将结果保存在变量 sumZ 中。

运行页面，调试信息如图 6.1 所示。

如图 6.1 所示，从第 09 行代码的输出结果来看，特殊值（NaN）与数值相加后结果仍为（NaN）；从第 13 行代码的输出结果来看，特殊值（Infinity）与数值相加后结果也仍为（Infinity）。

另外，特殊值（Infinity）和特殊值（Infinity）相加后，结果仍为特殊值（Infinity）；特殊值（-Infinity）和特殊值（-Infinity）相加后，结果仍为特殊值（-Infinity）；特殊值（-Infinity）和特

殊值（Infinity）相加后，结果变化为特殊值（NaN），表示不是一个有效数值。

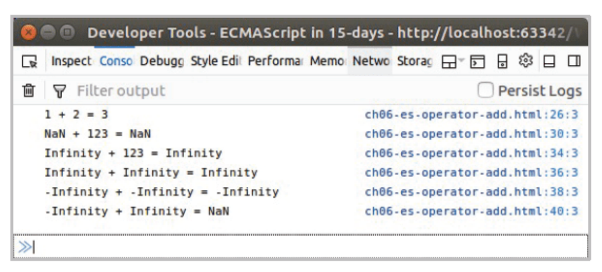


图 6.1　ECMAScript 加法（+）运算符

下面再看一个应用 ECMAScript 加法（+）运算符进行字符串连接操作的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-add-str.html 文件）。

【代码 6-2】

```
01  <script type="text/javascript">
02      var strHello = "Hello";
03      var strECMA = "ECMAScript";
04      console.log(strHello + " " + strECMA + "!");
05      var i = 1;
06      var strN = "8";
07      console.log(i + strN);
08      console.log(strN + i);
09  </script>
```

关于【代码 6-2】的分析如下：

第 02～04 行代码定义了两个字符串变量 strHello 和 strECMA。其中，在第 04 行代码中通过加法（+）运算符进行了字符串连接操作，并增加了空格和标点符号。

第 05～08 行代码通过加法（+）运算符对字符串和数值进行了连接操作。其中，第 05 行代码定义了变量 i，并初始化赋值为数值 1；第 06 行代码定义了变量 strN，并初始化赋值为字符串（"8"，为数值字符串）；第 07 行代码尝试通过加法（+）运算符将变量 i 与变量 strN 进行连接操作，具体来说就是将数值与字符串进行连接操作；第 08 行代码与第 07 行代码相反，尝试通过加法（+）运算符将变量 strN 与变量 i 进行连接操作，具体来说就是将字符串与数值进行连接操作。

运行页面，调试信息如图 6.2 所示。从第 04 行代码的输出结果来看，加法（+）运算符也可以用于字符串连接操作。另外，从第 07 和第 08 行代码的输出结果来看，对字符串与数值使用加法（+）运算符进行连接操作时，数值类型会被转换为字符串类型，然后进行字符串连接操作。即使字符串为数值类的字符串，参考第 06 行代码定义的变量 strN，也是进行字符串连接操作，而不会与数值进行算术运算。

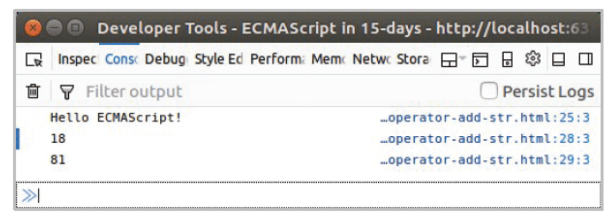


图 6.2　ECMAScript 加法（+）运算符连接字符串

6.1.3 减法运算符及表达式

ECMAScript 语法规范中规定减法运算符使用符号“－”来表示。减法（－）运算符除了可以进行正常的数值运算外，还支持对类似“NaN”和“Infinity”特殊值的运算。

下面看一个应用 ECMAScript 减法（－）运算符的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-minus.html 文件）。

【代码 6-3】

```
01  <script type="text/javascript">
02      var a = 2;
03      var b = 1;
04      var minusNumA = a - b;
05      console.log("2 - 1 = " + minusNumA);
06      var minusNumB = a.toString() - b;
07      console.log('"2" - 1 = ' + minusNumB);
08      var minusStr = "ECMAScript" - 123;
09      console.log('"ECMAScript" - 123 = ' + minusStr);
10      var c = NaN;
11      var d = 123;
12      var minusNaNA = c - d;
13      console.log("NaN - 123 = " + minusNaNA);
14      var minusNaNB = d - c;
15      console.log("123 - NaN = " + minusNaNB);
16      var x = Infinity;
17      var y = 321;
18      var minusInfinityA = x - y;
19      console.log("Infinity - 321 = " + minusInfinityA);
20      var minusInfinityB = y - x;
21      console.log("321 - Infinity = " + minusInfinityB);
22      var minusA = Infinity - Infinity;
23      console.log("Infinity - Infinity = " + minusA);
24      var minusB = -Infinity - -Infinity;
25      console.log("-Infinity - -Infinity = " + minusB);
26      var minusC = -Infinity - Infinity;
27      console.log("-Infinity - Infinity = " + minusC);
28      var minusD = Infinity - -Infinity;
29      console.log("Infinity - -Infinity = " + minusD);
30  </script>
```

关于【代码 6-3】的分析如下：

第 02～05 行代码定义了两个变量 a 和 b，并通过减法（－）运算符进行了算术运算。其中，在第 04 行代码中就是通过减法（－）运算符将变量 a 和变量 b 进行了算术相减运算。

第 06 和第 07 行代码在第 02～05 行代码的基础上做了一些修改，即先将变量 a 转换为字符串类型，再通过减法（－）运算符与变量 b 进行了算术相减运算。

第 08 和第 09 行代码通过减法（－）运算符对一个字符串（"EcmaScript"）和数值（123）进行了相减运算。

第 10～15 行代码通过减法（－）运算符对特殊值（NaN）进行了算术运算。其中，第 10 行代码定义了变量 c，并初始化赋值为特殊值（NaN）；第 12 行和第 14 行代码分别尝试将变量 c 与一

个具体数值进行减法运算。

第 16～21 行代码通过减法（－）运算符对特殊值（Infinity）进行了算术运算。其中，第 16 行代码定义了变量 x，并初始化赋值为特殊值（Infinity）；第 18 行和第 20 行代码分别尝试将变量 x 与一个具体数值进行相减运算。

第 22 和第 23 行代码通过减法（－）运算符对特殊值（Infinity）和特殊值（Infinity）进行了算术运算，并将结果保存在变量 minusA 中。

第 24 和第 25 行代码通过减法（－）运算符对特殊值（-Infinity）和特殊值（-Infinity）进行了算术运算，并将结果保存在变量 minusB 中。

第 26 和第 27 行代码通过减法（－）运算符对特殊值（-Infinity）和特殊值（Infinity）进行了算术运算，并将结果保存在变量 minusC 中。

第 28 和第 29 行代码通过减法（－）运算符对特殊值（Infinity）和特殊值（-Infinity）进行了算术运算，并将结果保存在变量 minusD 中。

运行页面，调试信息如图 6.3 所示。

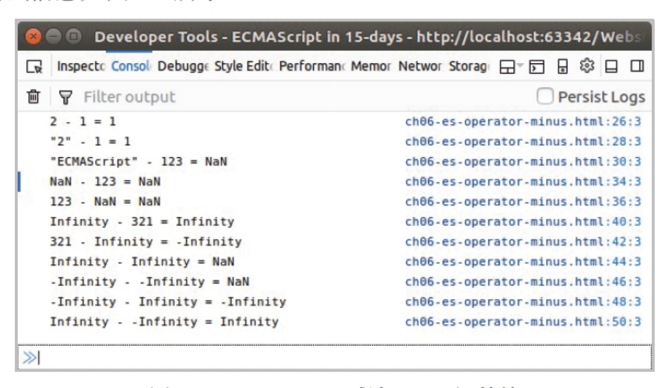


图 6.3　ECMAScript 减法（－）运算符

如图 6.3 所示，从第 07 行代码的输出结果来看，将数值转换为字符串后再执行减法（－）操作，仍可以得出正确的计算结果；从第 09 行代码的输出结果来看，对于非数值类的字符串与数值进行减法（－）操作，得出的结果为“NaN”。

从第 13 行和第 15 行代码的输出结果来看，特殊值（NaN）与数值执行减法（－）操作后，结果仍为（NaN）。

从第 19 行和第 21 行代码的输出结果来看，特殊值（Infinity）与数值执行减法（－）操作后，结果仍为（Infinity）。

从第 23 行代码输出的结果来看，特殊值（Infinity）和特殊值（Infinity）执行减法（－）操作后，结果为特殊值（NaN）。

从第 25 行代码输出的结果来看，特殊值（-Infinity）和特殊值（-Infinity）执行减法（－）操作后，结果仍为特殊值（NaN）。

从第 27 行代码输出的结果来看，特殊值（-Infinity）和特殊值（Infinity）执行减法（－）操作后，结果为特殊值（-Infinity）。

从第 29 行代码输出的结果来看，特殊值（Infinity）和特殊值（-Infinity）执行减法（－）操作后，结果为特殊值（Infinity）。

6.2 ECMAScript 乘性运算符及表达式

ECMAScript 语法规范中将乘法（*）、除法（/）和模数（%）运算符统一称为乘性运算符，用于执行数值之间的乘除和取余算术运算。

6.2.1 乘性运算符与表达式概述

ECMAScript 语法规范中定义的乘性运算符与表达式详见表 6.2。

表 6.2 ECMAScript 乘性运算符与表达式

运算符	描述	表达式示例	运算结果
*	乘	3*6	18
/	除	18/6	3
%	取余（模）数（保留整数）	5%3	2

6.2.2 乘法运算符及表达式

ECMAScript 语法规范中定义乘法运算符使用符号“*”来表示。乘法（*）运算符除了可以进行正常的数值运算外，还支持对类似“NaN”和“Infinity”特殊值的运算。

下面看一个应用乘法（*）运算符的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-multiple.html 文件）。

【代码 6-4】

```
01  <script type="text/javascript">
02      var a = 135;
03      var b = 246;
04      var mul = a * b;
05      console.log("135 * 246 = " + mul);
06      var e = 1e1001;
07      var mulE = e * e;
08      console.log("1e1001 * 1e1001 = " + mulE);
09      var mulEE = -e * e;
10      console.log("-1e1001 * 1e1001 = " + mulEE);
11      var c = NaN;
12      var d = 5;
13      var mulNaN = c * d;
14      console.log("NaN * 5 = " + mulNaN);
15      var x = Infinity;
16      var y = 0;
17      var mulInfinity = x * y;
18      console.log("Infinity * 0 = " + mulInfinity);
19      var z = 1;
20      var mulZ = x * z;
```

```
21          console.log("Infinity * 1 = " + mulZ);
22          var zz = -1;
23          var mulZZ = x * zz;
24          console.log("Infinity * (-1) = " + mulZZ);
25          var mulMul = Infinity * Infinity;
26          console.log("Infinity * Infinity = " + mulMul);
27      </script>
```

关于【代码 6-4】的分析如下：

第 02～05 行代码定义了两个变量 a 和 b，并通过乘法（*）运算符进行了算术运算。其中，在第 04 行代码中就是通过乘法（*）运算符将变量 a 和变量 b 进行了算术相乘运算。

第 06～10 行代码通过乘法（*）运算符进行了生成无穷大数的计算。其中，第 06 行代码定义了一个变量 e，并初始化为科学记数法数值（1e1001）；第 07 行和第 09 行代码使用乘法（*）运算符分别计算了数值（1e1001）的正负平方值。

第 11～14 行代码通过乘法（*）运算符分别对特殊值（NaN）进行了算术运算。其中，第 11 行代码定义了变量 c，并初始化赋值为特殊值（NaN）；第 13 行代码尝试将变量 c 与一个具体数值进行算术相乘运算。

第 15～24 行代码通过乘法（*）运算符对特殊值（Infinity）进行了算术运算。其中，第 15 行代码定义了变量 x，并初始化赋值为特殊值（Infinity）；第 17 行代码尝试将变量 x 与数值（0）进行算术相乘运算；第 20 行代码尝试将变量 x 与非零数值 1 进行算术相乘运算；第 23 行代码尝试将变量 x 与非零数值（-1）进行算术相乘运算。

第 25 和第 26 行代码通过乘法（*）运算符对特殊值（Infinity）和特殊值（Infinity）进行了算术运算。

运行页面，调试信息如图 6.4 所示。

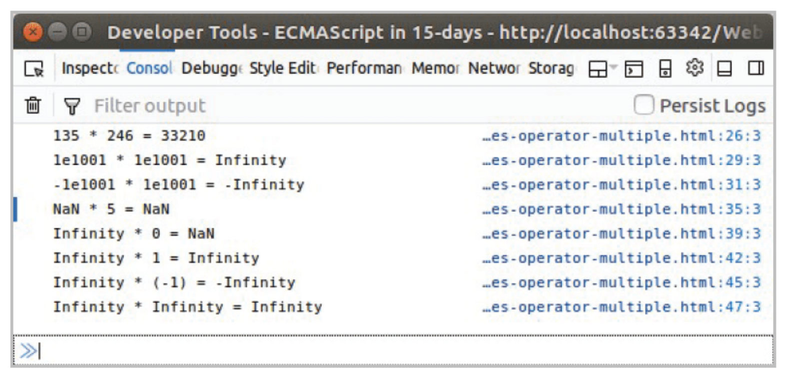


图 6.4　ECMAScript 乘法（*）运算符

如图 6.4 所示，从第 08 行和第 10 行代码的输出结果来看，如果乘法运算后结果太大或太小，生成的结果就是特殊值（Infinity 或-Infinity）。

从第 14 行代码的输出结果来看，特殊值（NaN）与数值相乘后结果为特殊值（NaN）。

从第 18 行代码的输出结果来看，特殊值（Infinity）与数值（0）相乘后结果为特殊值（NaN）。

从第 21 和第 24 行代码的输出结果来看，特殊值（Infinity）与非零数值（如 1 或-1）相乘后结果为特殊值（Infinity 或-Infinity）。

另外，如果特殊值（Infinity）与特殊值（Infinity）进行相乘运算后，结果仍为特殊值（Infinity）。

6.2.3 除法运算符及表达式

ECMAScript 语法规范中定义除法运算符使用符号“/”来表示。除法（/）运算符除了可以进行正常的数值运算外，还支持对类似“NaN”和“Infinity”特殊值的运算。

下面看一个应用除法（/）运算符的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-divide.html 文件）。

【代码 6-5】

```
01  <script type="text/javascript">
02      var i = 128;
03      var j = 16;
04      var sum = i / j;
05      console.log("128 / 16 = " + sum);
06      var a = 1e-1001;
07      var sumAA = 1 / a;
08      console.log("1 / 1e-1001 = " + sumAA);
09      var sumAB = -1 / a;
10      console.log("-1 / 1e-1001 = " + sumAB);
11      var p = NaN;
12      var q = 2;
13      var sumNaNA = p / q;
14      console.log("NaN / 2 = " + sumNaNA);
15      var sumNaNB = q / p;
16      console.log("2 / NaN = " + sumNaNB);
17      var sumZero = 2 / 0;
18      console.log("2 / 0 = " + sumZero);
19      var sumInfinityA = Infinity / 2;
20      console.log("Infinity / 2 = " + sumInfinityA);
21      var sumInfinityB = Infinity / (-2);
22      console.log("Infinity / -2 = " + sumInfinityB);
23      var sumInfinityZero = Infinity / 0;
24      console.log("Infinity / 0 = " + sumInfinityZero);
25      var sumSum = Infinity / Infinity;
26      console.log("Infinity / Infinity = " + sumSum);
27  </script>
```

关于【代码 6-5】的分析如下：

第 02～05 行代码定义了两个变量 i 和 j，并通过除法（/）运算符进行了算术运算。其中，在第 04 行代码中就是通过除法（/）运算符将变量 i 和变量 j 进行了算术除法运算。

第 06～10 行代码通过除法（/）运算符进行了无穷大的计算。其中，第 06 行代码定义了一个变量 a，并初始化为科学记数法数值（1e-1001）；第 07 和第 09 行代码使用除法（/）运算符分别计算了数值（正负 1）除以数值（1e-1001）的结果。

第 11～16 行代码通过除法（/）运算符分别对特殊值（NaN）进行了算术运算。其中，第 11 行代码定义了变量 p，并初始化赋值为特殊值（NaN）；第 13 行代码尝试将变量 p 与一个具体数值（2）进行算术除法运算；第 15 行代码尝试将一个具体数值（2）与变量 p 进行算术除法运算。

第 17 和第 18 行代码通过除法（/）运算符计算了数值（2）除以数值（0）的结果，也就是除

数为零的运算。

第 19～24 行代码通过除法（/）运算符对特殊值（Infinity）进行了算术运算。其中，第 19 行代码计算了特殊值（Infinity）除以非零数值（2）的结果；第 21 行代码计算了特殊值（Infinity）除以非零数值（-2）的结果；第 23 行代码计算了特殊值（Infinity）除以数值（0）的结果。

第 25 和第 26 行代码通过除法（/）运算符对特殊值（Infinity）和特殊值（Infinity）进行了算术除法运算，并将结果保存在变量 sumSum 中。

运行页面，调试信息如图 6.5 所示。

```
128 / 16 = 8                         ch06-es-operator-divide.html:26:3
1 / 1e-1001 = Infinity               ch06-es-operator-divide.html:29:3
-1 / 1e-1001 = -Infinity             ch06-es-operator-divide.html:31:3
NaN / 2 = NaN                        ch06-es-operator-divide.html:35:3
2 / NaN = NaN                        ch06-es-operator-divide.html:37:3
2 / 0 = Infinity                     ch06-es-operator-divide.html:39:3
Infinity / 2 = Infinity              ch06-es-operator-divide.html:41:3
Infinity / -2 = -Infinity            ch06-es-operator-divide.html:43:3
Infinity / 0 = Infinity              ch06-es-operator-divide.html:45:3
Infinity / Infinity = NaN            ch06-es-operator-divide.html:47:3
```

图 6.5　ECMAScript 除法（/）运算符

如图 6.5 所示，从第 08 和第 10 行代码的输出结果来看，如果除法运算后结果太大或太小，生成的结果就是特殊值（Infinity 或-Infinity）。

从第 14 和第 16 行代码的输出结果来看，特殊值（NaN）与数值进行除法运算后结果仍为特殊值（NaN）。

从第 18 行代码的输出结果来看，特殊值（Infinity）除以数值（0）后结果为特殊值（Infinity）。

从第 20 和第 22 行代码的输出结果来看，特殊值（Infinity）除以非零数值（如 2 或–2）后结果为特殊值（Infinity 或-Infinity）。

从第 24 行代码的输出结果来看，特殊值（Infinity）除以数值（0）后结果为特殊值（Infinity）。

另外，如果执行特殊值（Infinity）除以特殊值（Infinity）的算术运算，则其结果为特殊值（NaN）。

6.2.4　取模运算符及表达式

ECMAScript 语法规范中定义取模运算符使用百分数符号“%”来表示。取模（%）运算符除了可以进行正常的数值运算外，还支持对类似“NaN”和“Infinity”特殊值的运算。

下面看一个应用取模（%）运算符的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-mod.html 文件）。

【代码 6-6】

```
01  <script type="text/javascript">
02      var a = 5;
03      var b = 3;
04      var mod = a % b;
```

```
05        console.log("5 % 3 = " + mod);
06        var modZeroDiv = 0 % 2;
07        console.log("0 % 2 = " + modZeroDiv);
08        var modDivZero = 2 % 0;
09        console.log("2 % 0 = " + modDivZero);
10        var modInfinityDiv = Infinity % 2;
11        console.log("Infinity % 2 = " + modInfinityDiv);
12        var modInfinityInfinity = Infinity % Infinity;
13        console.log("Infinity % Infinity = " + modInfinityInfinity);
14        var modDivInfinity = 2 % Infinity;
15        console.log("2 % Infinity = " + modDivInfinity);
16    </script>
```

关于【代码 6-6】的分析如下：

第 02～05 行代码定义了两个变量 a 和 b，并通过取模（%）运算符进行了算术运算。其中，在第 04 行代码中就是通过取模（%）运算符将变量 a 和变量 b 进行了算术取模运算。

第 06 和第 07 行代码通过取模（%）运算符对被除数为数值（0）、除数为非零数值进行取模运算。

类似的，第 08 和第 09 行代码通过取模（%）运算符对除数为数值（0）、被除数为非零数值进行取模运算。

第 10 和第 11 行代码通过取模（%）运算符对被除数为特殊值（Infinity）的情况进行了取模运算。

第 12 和第 13 行代码通过取模（%）运算符对被除数和除数均为特殊值（Infinity）的情况进行了取模运算。

第 14～15 行代码通过取模（%）运算符对除数为特殊值（Infinity）的情况进行了取模运算。

运行页面，调试信息如图 6.6 所示。

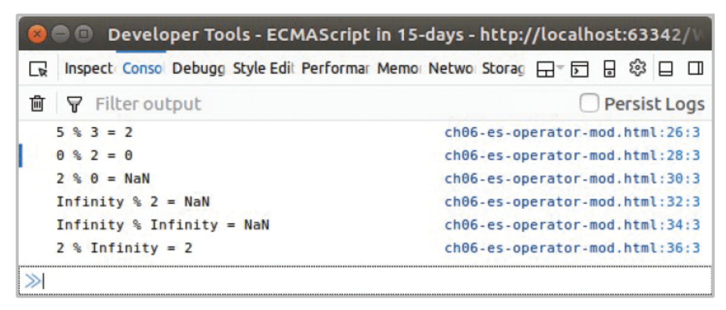


图 6.6　ECMAScript 取模（%）运算符

如图 6.6 所示，从第 05 行代码的输出结果来看，表达式（5 % 3 = ）的运算结果为数值（2），与取余数的结果一致。

从第 07 行代码的输出结果来看，表达式（0 % 2 = ）的运算结果为数值（0），与取余数的结果一致。

从第 09 行代码的输出结果来看，表达式（2 % 0 = ）的运算结果为特殊值（NaN），因为除数为零的取模运算是无意义的，所以结果为特殊值（NaN）。

从第 11 行代码的输出结果来看，表达式（Infinity % 2 = ）的运算结果为特殊值（NaN），因为对特殊值（Infinity）进行取模运算也是无意义的，所以结果为特殊值（NaN）。

从第 13 行代码的输出结果来看，表达式（Infinity % Infinity = ）的运算结果为特殊值（NaN），说明特殊值（Infinity）对特殊值（Infinity）取模也是没有意义的。

从第 15 行代码的输出结果来看，表达式（2 % Infinity = ）的运算结果为数值（2），这就说明对任意数值使用除数为特殊值（Infinity）进行取模运算，结果均为被除数的数值本身。

6.3　ECMAScript 一元运算符及表达式

ECMAScript 语法规范中对只有一个参数（对象或值）进行操作的运算符，统称为一元运算符。一元运算符包括 new、delete、void、增减量运算、一元加减法等，下面对这些一元运算符逐一进行介绍。

6.3.1　一元运算符与表达式概述

ECMAScript 语法规范中定义的一元运算符与表达式详见表 6.3。

表 6.3　ECMAScript 一元运算符与表达式

运算符	描述	表达式示例	运算结果
new	创建或新建对象	new Object	/
delete	清除或删除对已存在对象的属性或方法的引用	delete obj.prop	/
void	对任何值返回 undefined	/	undefined
++	增量运算	++1、1++	/
--	减量运算	--1、1--	/
+	一元加法运算	+1	1
-	一元减法运算	-1	-1

6.3.2　new 和 delete 运算符及表达式

对于 new 和 delete 运算符，相信学习过 C 语言或 Java 语言的读者一定不陌生。ECMAScript 语法规范中规定 new 运算符用于创建新的对象，而 delete 运算符用于删除对已存在对象的属性或方法的引用。

这里需要注意的是，ECMAScript 脚本语言与 C 语言和 Java 语言略有不同，ECMAScript 语法规范中没有“类”的概念，只有“对象”的概念。其实，ECMAScript 中的对象与 C 和 Java 语言中的类，在含义、功能和用法上基本类同。读者把 ECMAScript 脚本语言中的对象当作 C 语言和 Java 语言中的类来理解，是完全没有问题的。

下面看一个应用 new 和 delete 运算符的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-new-delete.html 文件）。

【代码 6-7】

```
01    <script type="text/javascript">
```

```
02        // obj test
03        console.log("delete obj :");
04        var obj = new Object;
05        obj.userid = "M10";
06        obj.username = "Leo Messi";
07        obj.gentle = "male";
08        console.log("userid : " + obj.userid);
09        console.log("username : " + obj.username);
10        console.log("gentle : " + obj.gentle);
11        console.log(obj);
12        delete obj.gentle;
13        console.log(obj);
14        delete obj.username;
15        console.log(obj);
16        delete obj.userid;
17        console.log(obj);
18        obj = null;
19        // obj2 test
20        console.log("delete obj2 :");
21        var obj2 = new Object;
22        obj2.userid = "M10";
23        obj2.username = "Leo Messi";
24        obj2.gentle = "male";
25        console.log(obj2);
26        delete obj2;
27        console.log(obj2);
28        obj2 = null;
29        console.log(obj2);
30    </script>
```

关于【代码 6-7】的分析如下：

第 04 行代码通过 new 运算符创建了一个对象 obj（var obj = new Object;），并在第 05～07 行代码中为该对象 obj 定义了 3 个属性，同时初始化了具体属性值。

第 08～10 行代码分别通过对象 obj 的 3 个属性输出了其属性值，第 11 行代码则是直接通过对象 obj 输出了其全部属性值。

第 12、第 14 和第 16 行代码依次通过 delete 运算符删除了对象 obj 的 3 个属性，同时第 13、第 15 和第 17 行代码在每次 delete 运算后，输出了对象 obj 的全部属性值。

第 21～29 行代码与前面的代码类似，不同之处在于第 26 行代码中，我们尝试直接通过 delete 运算符删除对象 obj2。

运行页面，调试信息如图 6.7 所示。

如图 6.7 所示，从第 08～10 行代码的输出结果来看，与我们初始化定义的属性值是一致的。

从第 11 行代码输出的结果来看，直接输出对象 obj 的效果是一个以 Object 开头的 JSON 数据格式，内部包含全部属性名称及其对应的属性值。

从第 13、第 15 和第 17 行代码输出的结果来看，每次经过 delete 运算后，对象 obj 的属性均被成功删除，直至最后输出了一个空的 Object 对象。

图 6.7　ECMAScript 一元运算符（new & delete）

从第 25～27 行代码输出的结果来看，第 26 行代码中尝试通过 delete 运算直接删除对象 obj2 的操作没有效果，这是为什么呢？让我们返回关于 delete 运算符定义的描述：“delete 运算符用于删除对已存在对象的属性或方法的引用”，这就说明 delete 运算符只对于对象的属性或方法有效，而无法直接删除对象本身。

那么如何删除对象呢？我们看第 18 行和第 28 行代码中定义，通过直接给对象赋值 null，就可以清空对象了，从第 29 行代码输出的结果就可以看出来。

但是，清空对象并不意味着对象就从内存中被释放了，ECMAScript 脚本语言有专门的垃圾回收机制负责内存释放的操作。

6.3.3　void 运算符及表达式

ECMAScript 语法规范中定义 void 运算符对任何值均返回（undefined）值，void 运算符的作用是避免输出不应该输出的值。

例如，在 HTML 页面的<a>标签内调用 ECMAScript（JavaScript）函数时，若打算正确实现该功能，则函数一定不能返回有效值，否则浏览器页面就会被清空，仅仅会替代显示出函数的返回结果。

下面先看一个未应用 void 运算符的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-void.html 文件）。

【代码 6-8】

```
01  <p>
02      无 void 操作符测试:<br>
03      <a href="javascript:window.open('about:blank')">无 void 操作符</a><br>
04  </p>
```

关于【代码 6-8】的分析如下：

这段代码的主要目的就是调用 window.open()方法打开一个空的页面，但因为 window.open()方法会有一个返回值（即对新打开窗口的引用），所以该页面会显示该返回值。

运行页面，调试信息如图 6.8 所示。单击页面中箭头所指的“无 void 操作符”超链接，页面效果如图 6.9 所示。因为 window.open()方法会有一个返回值，即对新打开窗口的引用（一个 window 对象），所以原页面中的内容已被该 window 对象的引用值[object Window]强行替换掉了。

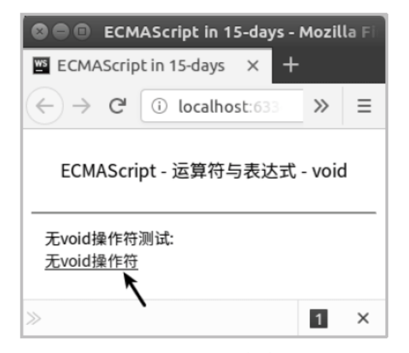


图 6.8　ECMAScript 一元运算符（无 void）（1）



图 6.9　ECMAScript 一元运算符（无 void）（2）

那么如何避免该问题的出现呢？这个时候 void 运算符就可以发挥作用了。

下面继续看一个应用 void 运算符的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-void.html 文件）。

【代码 6-9】

```
01    <p>
02    有 void 操作符测试:<br>
03    <a href="javascript:void(window.open('about:blank'))">有 void 操作符</a><br>
04    </p>
```

关于【代码 6-9】的分析如下：

这段代码的主要目的同样是调用 window.open()方法打开一个空的页面，但与【代码 6-8】的处理方式不同的是，在 window.open()方法外面通过 void 运算符将其包裹在其中了，所以就将返回值强制转换为 undefined 值了。

运行 HTML 页面，并使用浏览器控制台查看调试信息，页面初始效果如图 6.10 所示。单击页面中箭头所指的“有 void 操作符”超链接，页面效果如图 6.11 所示。虽然 window.open()方法会有一个返回值，但是被 void 运算符强制转换为 undefined 值，而 undefined 值是无意义的，因此就会保留原始的内容。

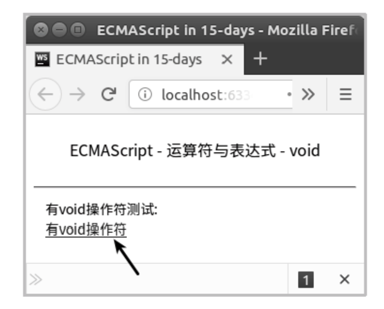


图 6.10　ECMAScript 一元运算符（有 void）（1）

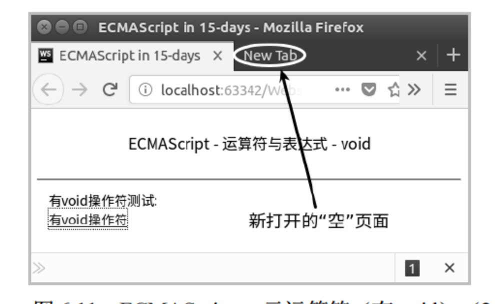


图 6.11　ECMAScript 一元运算符（有 void）（2）

6.3.4　前增量与前减量运算符及表达式

对于前增量与前减量运算符，相信学习过 C 语言或 Java 语言的读者也一定很熟悉。ECMAScript 语法规范中同样定义了前增量与前减量这两个运算符，使用方法与 C 语言和 Java 语言是一致的。

所谓前增量运算符，就是在变量前放两个加号（++），功能是在数值上加 1；同理，前减量运算符就是在变量前放两个减号（−−），功能是在数值上减 1。注意，两个加号（++）或两个减号（−−）放在变量前与放在变量后，在功能上是有区别的，后面我们会详细介绍。

下面看一个应用前增量与前减量运算符的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-pre-ppmm.html 文件）。

【代码 6-10】

```
01  <script type="text/javascript">
02      var i = 1;
03      console.log("i = " + i);
04      ++i;
05      console.log("++i, i = " + i);
06      console.log("++i = " + ++i);
07      console.log("i = " + i);
08      var j = 3;
09      console.log("j = " + j);
10      --j;
11      console.log("--j, j = " + j);
12      console.log("--j = " + --j);
13      console.log("j = " + j);
14  </script>
```

关于【代码 6-10】的分析如下：

第 02 行代码定义了第一个变量 i，并初始化为数值 1。

第 04 行代码使用前增量运算符对变量 i 进行了操作（++i）。

第 06 行代码在浏览器控制台中输出了表达式（++i）的数值，这里是对变量 i 使用前增量运算符后的结果。

第 08 行代码定义了第二个变量 j，并初始化为数值 3。

第 10 行代码使用前减量运算符对变量 j 进行了操作（−−j）。

第 12 行代码在浏览器控制台中输出了表达式（−−j）的数值，这里是对变量 j 使用前减量运算符后的结果。

第 13 行代码再次在浏览器控制台中输出了变量 j 的数值。

运行页面，调试信息如图 6.12 所示。

如图 6.12 所示，在第 04 行代码对变量 i 使用前增量运算符操作（++i）后，第 05 行代码输出的变量 i 的值为 2，说明前增量运算符对变量 i 的操作成功了。

从第 06 行代码输出的数值结果 3 来看，说明直接输出表达式（++i）的数值为 3。

从第 07 行代码输出的数值结果 3 来看，经过第 06 行代码中前增量运算符对变量 i 的操作后，变量 i 的数值也变为 3。

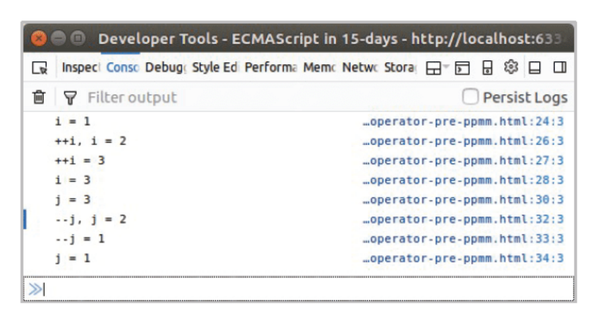


图 6.12 ECMAScript 一元运算符（前增量与前减量）

类似地，在第 10 行代码对变量 j 使用前减量运算符操作（−−j）后，第 11 行代码输出的变量 j 的值为 2，说明前减量运算符对变量 j 的操作成功了。

从第 12 行代码输出的数值结果 1 来看，说明直接输出表达式（−−j）的数值为 1。

从第 13 行代码输出的数值结果 1 来看，经过第 12 行代码中前减量运算符对变量 j 的操作后，变量 j 的数值也变为 1。

以上就是对前增量与前减量运算符的测试过程，下面介绍对这两个运算符对应的后增量与后减量运算符。

6.3.5 后增量与后减量运算符及表达式

对于后增量与后减量运算符，可以理解为是相对于前增量与前减量运算符而言的。所谓后增量运算符，就是在变量后放两个加号（++），功能是在数值上加 1；同理，后减量运算符就是在变量后放两个减号（−−），功能是在数值上减 1。如前面所述，两个加号（++）或两个减号（−−）放在变量后与放在变量前，在功能上是有区别的。下面我们就通过具体代码实例来详细分析。

下面是一个应用后增量与后减量运算符的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-suf-ppmm.html 文件），该代码示例是在【代码 6-10】的基础上修改而成的。

【代码 6-11】

```
01  <script type="text/javascript">
02      var i = 1;
03      console.log("i = " + i);
04      i++;
05      console.log("i++, i = " + i);
06      console.log("i++ = " + i++);
07      console.log("i = " + i);
08      var j = 3;
09      console.log("j = " + j);
10      j--;
11      console.log("j--, j = " + j);
12      console.log("j-- = " + j--);
13      console.log("j = " + j);
14  </script>
```

关于【代码 6-11】的分析如下：

第 02 行代码定义了第一个变量 i，并初始化为数值 1。

第 04 行代码使用后增量运算符对变量 i 进行了操作（i++）。

第 06 行代码在浏览器控制台中输出了表达式（i++）的数值，这里是对变量 i 使用后增量运算符后的结果。

第 08 行代码定义了第二个变量 j，并初始化为数值 3。

第 10 行代码使用后减量运算符对变量 j 进行了操作（j−−）。

第 12 行代码在浏览器控制台中输出了表达式（j−−）的数值，这里是对变量 j 使用后减量运算符后的结果。

第 13 行代码再次在浏览器控制台中输出了变量 j 的数值。

运行页面，调试信息如图 6.13 所示。

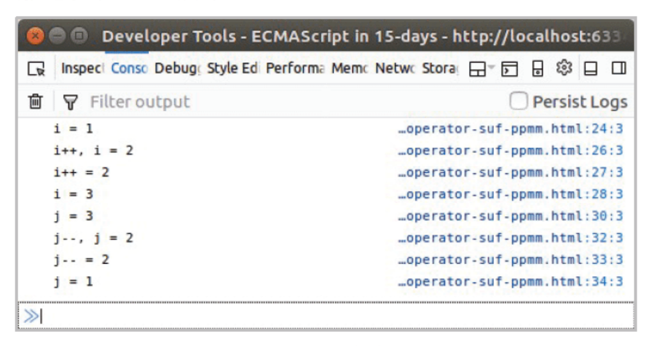


图 6.13　ECMAScript 一元运算符（后增量与后减量）

如图 6.13 所示，在第 04 行代码对变量 i 使用后增量运算符操作（i++）后，第 05 行代码输出的变量 i 的值为 2，说明后增量运算符对变量 i 的操作成功了。

从第 06 行代码输出的数值结果 2 来看，说明直接输出表达式（i++）的数值仍为 2，这就与前增量运算符有区别了。

从第 07 行代码输出的数值结果 3 来看，经过第 06 行代码中后增量运算符对变量 i 的操作后，变量 i 的数值确实是变为 3 了。

类似的，在第 10 行代码对变量 j 使用后减量运算符操作（j−−）后，第 11 行代码输出的变量 j 的值为 2，说明后减量运算符对变量 j 的操作成功了。

从第 12 行代码输出的数值结果 2 来看，说明直接输出表达式（j−−）的数值仍为 2，这就与前减量运算符有区别了。

从第 13 行代码输出的数值结果 1 来看，经过第 12 行代码中后减量运算符对变量 j 的操作后，变量 j 的数值确实是变为 1 了。

以上就是对后增量与后减量运算符的测试过程，读者可以仔细对比【代码 6-10】与【代码 6-11】中运算符使用的特点。

6.3.6　一元加法与一元减法运算符及表达式

ECMAScript 语法规范中定义的一元加法与一元减法运算符，与我们在数学中学习到的定义基本一致，但在功能用法上又有所增强。

所谓一元加法运算符，就是在变量前放一个加号（+），其对数值基本是无意义的（因为对数

值使用一元加法运算符后，数值仍是其本身），但其可以将字符串转换为数值，还可以将十六进制数转换为十进制数。同理，一元减法运算符就是在变量前放一个减号（–），其可对数值求相反数，同时对字符串和十六进制数也起作用。

下面看一个应用一元加法与一元减法运算符的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-uni-add-minus.html 文件）。

【代码 6-12】

```
01  <script type="text/javascript">
02      var i = 123;
03      console.log("+i = " + +i);
04      console.log("-i = " + -i);
05      var j = -123;
06      console.log("+j = " + +j);
07      console.log("-j = " + -j);
08      var str1 = "123";
09      console.log("typeof str1 = " + typeof str1);
10      console.log("+str1 = " + +str1);
11      console.log("typeof +str1 = " + typeof +str1);
12      var str2 = "-123";
13      console.log("-str2 = " + -str2);
14      var ix16 = 0xff;
15      console.log("+ix16 = " + +ix16);
16      console.log("+ix16 = " + -ix16);
17  </script>
```

关于【代码 6-12】的分析如下：

第 02 行代码定义了第一个变量 i，并初始化为正数值 123。

第 03 行代码使用一元加法运算符对变量 i 进行了操作（+i）。

第 04 行代码使用一元减法运算符对变量 i 进行了操作（-i）。

第 05 行代码定义了第二个变量 j，并初始化为负数值-123。

第 06 行代码使用一元加法运算符对变量 j 进行了操作（+j）。

第 07 行代码使用一元减法运算符对变量 j 进行了操作（-j）。

第 08 行代码定义了第三个变量 str1，并初始化为正数形式的字符串"123"。

第 09 行代码使用 typeof 运算符对变量 str1 进行了操作（typeof str1）。

第 10 行代码使用一元加法运算符对变量 str1 进行了操作（+str1）。

第 11 行代码在第 10 行代码的基础上，使用 typeof 运算符和一元加法运算符对变量 str1 进行了操作（typeof +str1）。

第 12 行代码定义了第四个变量 str2，并初始化为负数形式的字符串"-123"。

第 13 行代码使用一元减法运算符对变量 str2 进行了操作（-str2）。

第 14 行代码定义了第五个变量 ix16，并初始化为十六进制形式的字符串 0xff。

第 15 行代码使用一元加法运算符对变量 ix16 进行了操作（+ix16）。

第 16 行代码使用一元减法运算符对变量 ix16 进行了操作（-ix16）。

运行页面，调试信息如图 6.14 所示。

如图 6.14 所示，从第 03 行代码输出的数值结果 123 来看，说明一元加法运算符对正数值是没有意义的。那么一元加法运算符对负数值呢？从第 06 行代码输出的结果-123 来看，一元加法运算

符对负数值同样没有意义，这就说明一元加法运算对数值都没有意义。

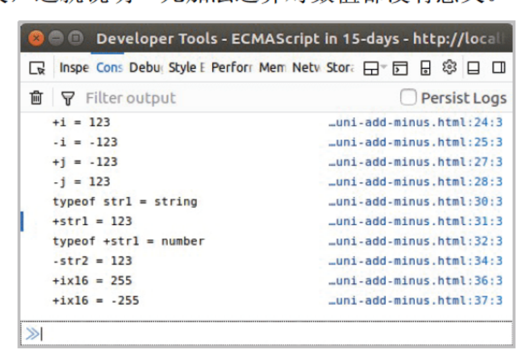


图 6.14　ECMAScript 一元运算符（一元加法与一元减法）

从第 04 行代码输出的数值结果-123 来看，说明一元减法运算符起到了求负的作用，将正数 123 转换为了负数-123。从第 07 行代码输出的数值结果 123 来看，说明一元减法运算符同样起到了求负的作用，将负数-123 转换为了正数 123。

从第 09 行代码输出的结果 string 来看，第 08 行代码定义的变量 str1 是一个字符串变量。

从第 10 行代码输出的结果 123 来看，对变量 str1 使用一元加法运算符后，将字符串类型转换为了数值类型，这从第 11 行代码输出的结果 number 可以得到确认。

从第 13 行代码输出的结果 123 来看，对字符串变量 str2 使用一元减法运算符后，不但可以将字符串类型转换为数值类型，而且还能对数值求负。

从第 15 和第 16 行代码输出的结果 255 和-255 来看，一元加法和一元减法运算符还可以将十六进制数转换成十进制数，同时一元减法运算符仍有求负功能。

6.4　ECMAScript 关系运算符及表达式

在 ECMAScript 语法规范中对两个数值执行比较运算的运算符，统称为关系运算符。关系运算符一般包括小于、大于、小于等于和大于等于 4 种。关系运算符表达式均会返回一个布尔值。

6.4.1　关系运算符与表达式概述

ECMAScript 语法规范中定义的关系运算符与表达式详见表 6.4。

表 6.4　ECMAScript 关系运算符与表达式

运算符	描述	示例	结果	前提条件
>	大于	x>0	false	x=0
>=	大于等于	x>=0	true	x=0
<	小于	x<0	false	x=0
<=	小于等于	x<=0	true	x=0

6.4.2 数值关系运算符表达式

在 ECMAScript 语法规范中，关系运算符主要应用于数值的比较运算。对于每一个数值关系运算符表达式均会返回一个布尔值，通过判断该布尔值来判断数值比较的结果。

下面看一个数值关系运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-relation-number.html 文件）。

【代码 6-13】

```
01  <script type="text/javascript">
02      var bR_1 = 2 > 1;
03      console.log("2 > 1 = " + bR_1);
04      var bR_2 = 2 < 1;
05      console.log("2 < 1 = " + bR_2);
06      var bR_3 = 2 >= 1;
07      console.log("2 >= 1 = " + bR_3);
08      var bR_4 = 2 <= 1;
09      console.log("2 <= 1 = " + bR_4);
10  </script>
```

关于【代码 6-13】的分析如下：

这段代码主要就是对数值 1 和 2 分别使用大于（>）、小于（<）、大于等于（>=）和小于等于（<=）这 4 种关系运算符进行了比较运算。

运行页面，调试信息如图 6.15 所示。数值关系运算符表达式输出的结果完全符合我们的预期。

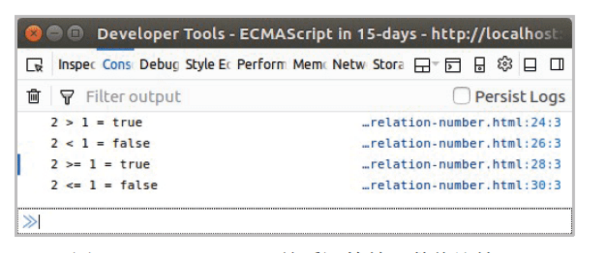


图 6.15 ECMAScript 关系运算符（数值比较）

6.4.3 字符串关系运算符表达式

在 ECMAScript 语法规范中，关系运算符还可以应用于字符串的比较运算，这也是计算机编程语言的一大特点。对于字符串也可以进行比较运算，似乎超出了我们知识范畴。不过对于学习过 C 语言或 Java 语言的读者来说，字符串比较运算并不陌生，这是因为在计算机编程语言中，关系运算符比较的是字符在计算机中的编码（ASCII 编码）。

下面看一个字符串关系运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-relation-string.html 文件）。

【代码 6-14】

```
01  <script type="text/javascript">
02      var bR_str_1 = "A" > "B";
```

```
03          console.log("A > B = " + bR_str_1);
04          var bR_str_2 = "a" < "b";
05          console.log("a < b = " + bR_str_2);
06          var bR_str_3 = "A" > "a";
07          console.log("A > a = " + bR_str_3);
08          var bR_str_4 = "apple" < "banana";
09          console.log("apple < banana = " + bR_str_4);
10          var bR_str_5 = "apple" > "Banana";
11          console.log("apple < Banana = " + bR_str_5);
12      </script>
```

关于【代码 6-14】的分析如下：

这段代码主要就是对字符串分别使用大于（>）和小于（<）关系运算符进行了比较运算。

运行页面，调试信息如图 6.16 所示。

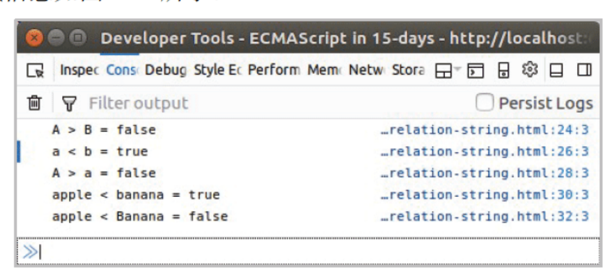


图 6.16　ECMAScript 关系运算符（字符串比较）

如图 6.16 所示，在尝试对大写字母 A 和 B 进行了比较运算"A" > "B"后，关系运算符表达式"A" > "B"的返回值为 false，说明大写字母 A 小于大写字母 B（比较 ASCII 编码）。

在尝试对小写字母 a 和 b 进行了比较运算"a" < "b"后，关系运算符表达式"a" < "b"的返回值为 true，说明小写字母 a 小于小写字母 b（比较 ASCII 编码）。

在尝试对大写字母 A 和小写字母 a 进行了比较运算"A" > "a"后，关系运算符表达式"A" > "a"的返回值为 false，说明大写字母 A 小于小写字母 a（比较 ASCII 编码），而不是大写字母 A 等于小写字母 a。

在尝试对小写单词 apple 和小写单词 banana 进行了比较运算"apple" < "banana"后，关系运算符表达式"apple" < "banana"的返回值为 true，说明小写单词 apple 的首字母 a 小于小写单词 banana 的首字母 b（比较 ASCII 编码）。

在尝试对小写单词 apple 和大写单词 Banana 进行了比较运算"apple" < "Banana"后，关系运算符表达式"apple" < "Banana"的返回值为 false，说明小写单词 apple 的首字母 a 大于大写单词 Banana 的首字母 B（比较 ASCII 编码）。

6.4.4　数值与字符串关系运算符表达式

在 ECMAScript 语法规范中，既然关系运算符可以应用于字符串的比较运算，那么对于数值和字符串的比较运算也是支持的。

下面看一个字符串关系运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-relation-num-str.html 文件）。

【代码 6-15】

```
01  <script type="text/javascript">
02      var bR_num_str_1 = "11" > "2";
03      console.log("'11' > '2' = " + bR_num_str_1);
04      var bR_num_str_2 = "11" > 2;
05      console.log("'11' > 2 = " + bR_num_str_2);
06      var bR_num_str_3 = "a" > 2;
07      console.log("'a' > 2 = " + bR_num_str_3);
08      var bR_num_str_4 = "a" <= 2;
09      console.log("'a' <= 2 = " + bR_num_str_4);
10  </script>
```

关于【代码 6-15】的分析如下：

这段代码主要就是对数值和字符串分别使用大于（>）和小于等于（<=）关系运算符进行了比较运算。

运行页面，调试信息如图 6.17 所示。

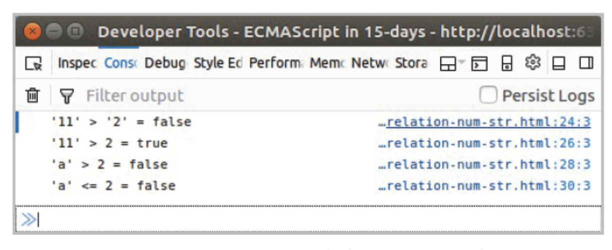


图 6.17　ECMAScript 关系运算符（数值与字符串比较）

如图 6.17 所示，在尝试对数值型字符串 11 和字符串 2 进行了比较运算"11" ＞ "2"后，关系运算符表达式"11" > "2"的返回值为 false，说明字符串 11 的首字符 1 是小于字符串 2 的（比较 ASCII 编码）。

在尝试对数值型字符串 “11” 和数值 2 进行了比较运算"11" > 2 后，关系运算符表达式"11" > 2 的返回值为 true，这是因为 ECMAScript 语法规范中定义了当字符串与数值进行关系运算时，字符串会先转换为数值，再与数值进行比较运算。

在尝试对数值型字符串 a 和数值 2 进行了比较运算"a" > 2 后，关系运算符表达式"a" > 2 的返回值为 false，这是因为字符串 a 转换为数值后返回值是 NaN。

在尝试对数值型字符串 a 和数值 2 进行了比较运算"a" <= 2 后，关系运算符表达式"a" <= 2 的返回值为 false，这同样是因为字符串 a 转换为数值后返回值是 NaN。

对于原始值 NaN 与数值进行关系运算后，返回的结果均是 false，这也正是 ECMAScript 语法规范中所定义的。

6.5　ECMAScript 等性运算符及表达式

在 ECMAScript 语法规范中判断两个变量是否相等的运算符，统称为等性运算符。等性运算符一般包括等号、不等号、严格相等和非严格相等，前两者用于处理原始值，后两者用于处理对象。

等性运算符表达式均会返回一个布尔值。

6.5.1　等性运算符与表达式概述

ECMAScript 语法规范中定义的等性运算符与表达式详见表 6.5。

表 6.5　ECMAScript 等性运算符与表达式

运算符	描述	示例	结果	前提条件
==	等号	x==1	true	x=1
!=	不等号	x!=1	false	x=1
===	严格相等（对象相等）	x===1	false	x="1"
!==	非严格相等（对象不相等）	x!==1	true	x="1"

6.5.2　等号与不等号运算符表达式

在 ECMAScript 语法规范定义中，等号是由两个等于号（==）来表示的，其含义是当两个运算数相等时返回布尔值 true，否则返回布尔值 false；不等号是由感叹号（!）和等于号（=）的组合（!=）来表示的，其含义是当两个运算数不相等时返回布尔值 true，否则返回布尔值 false。

为了确定两个运算数是否相等，运算前均会对这两个运算数进行类型转换。执行类型转换的基本规则如下：

- 如果一个运算数是 Boolean 值，在比较是否相等之前，会将其转换为相应数值，具体是 false 转换成 0、true 转换为 1。
- 如果一个运算数是字符串（一般指数值型字符串，如"123"），另一个是数字，在检查相等性之前，会将其转换为数值。
- 如果一个运算数是对象，另一个是字符串，在检查相等性之前，会将该对象转换为字符串。
- 如果一个运算数是对象，另一个是数值，在检查相等性之前，会将该对象转换为数值。

在比较运算时，等号与不等号运算符还会遵循以下几项规则：

- 原始值 null 和 undefined 是相等的。
- 在相等性运算时，不能把原始值 null 和 undefined 转换为其他值。
- 如果某个运算数是 NaN，那么等号将返回 false，不等号将返回 true。
- 如果两个运算数都是对象，那么比较的是各自的引用值。
- 如果两个运算数指向同一对象，那么等号运算会返回 true。

下面看一个等号与不等号运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-equal.html 文件）。

【代码 6-16】

```
01  <script type="text/javascript">
02      var bR_1_1 = 1 == 2;
03      console.log("1 == 2 = " + bR_1_1);
```

```
04          var bR_1_0 = 1 != 2;
05          console.log("1 != 2 = " + bR_1_0);
06          var bR_2_0 = false == 0;
07          console.log("false == 0 = " + bR_2_0);
08          var bR_2_1 = true == 1;
09          console.log("true == 1 = " + bR_2_1);
10          var bR_2_2 = true == 2;
11          console.log("true == 2 = " + bR_2_2);
12          var bR_3 = "1" == 1;
13          console.log("'1' == 1 = " + bR_3);
14          var bR_4 = null == 0;
15          console.log("null == 0 = " + bR_4);
16          var bR_5 = undefined == 0;
17          console.log("undefined == 0 = " + bR_5);
18          var bR_6 = undefined == null;
19          console.log("undefined == null = " + bR_6);
20          var bR_7 = NaN == 1;
21          console.log("NaN == 1 = " + bR_7);
22          var bR_8 = NaN == NaN;
23          console.log("NaN == NaN = " + bR_8);
24          var bR_9 = NaN != NaN;
25          console.log("NaN != NaN = " + bR_9);
26          var bR_10 = "NaN" == NaN;
27          console.log("'NaN' == NaN = " + bR_10);
28      </script>
```

关于【代码 6-16】的分析如下：

这段代码主要就是对数值、字符串、null、undefined 和 NaN 分别使用等号（==）和不等号（!=）运算符进行了比较运算，具体包括数值与数值、数值与字符串、null 与 undefined、NaN 与数值及 NaN 与自身等。

运行页面，调试信息如图 6.18 所示。

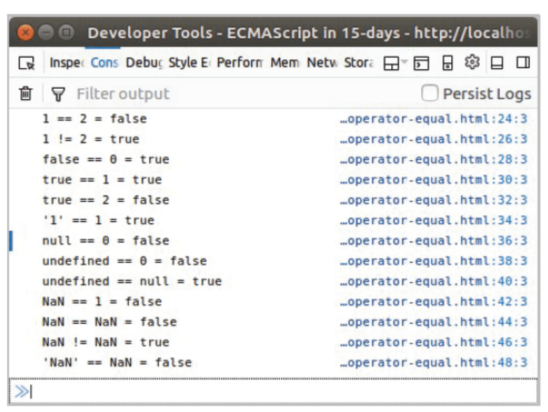


图 6.18　ECMAScript 等性运算符（等号与不等号）

如图 6.18 所示，从第 03 行和第 05 行代码输出的结果来看，等号和不等号运算符可用于判断两个运算数是否相等。

从第 07 和第 09 行代码输出的结果来看，在对 true 和 false 进行等性运算时，true 会先转换为数值 1，false 会先转换为数值 0，然后进行比较。因此，第 07 和第 09 行代码输出的结果为 true，

第 11 行代码输出的结果为 false。

从第 13 行代码输出的结果来看，在对字符串和数值进行等性运算时，字符串会先转换为数值，再与另一个数值进行比较。因此，第 13 行代码输出的结果为 true。

从第 15 和第 17 行代码输出的结果来看，null 和 undefined 与数值进行等性运算时，这两个原始值是不能转换为数值的。从第 19 行代码输出的结果来看，在对 null 和 undefined 进行等性运算时（null == undefined），返回的结果为 true，也就是说 null 和 undefined 的值是相等的。

从第 21、第 23、第 25 和第 27 行代码输出的结果来看，原始值 NaN 不但与数值进行等性运算时是不相等的，且与自身进行等性运算时（NaN == NaN、"NaN" == NaN）也是不相等的。

6.5.3　严格相等与非严格相等运算符表达式

在 ECMAScript 语法规范中，严格相等和非严格相等与相等和不等是同类的运算符，只不过严格相等和非严格相等的不同之处在于，进行比较运算之前不会对两个运算数进行类型转换。

ECMAScript 语法规范中定义严格相等使用三个等于号（===）来表示，其只有在无须类型转换运算数就相等的条件下，返回值才会为 true。非严格相等使用感叹号（!）和两个等于号（==）的组合（!==）来表示的，其只有在无须类型转换运算数就不相等的条件下，返回值才会为 true。

通过对上一小节的学习，我们知道等号与不等号运算时会对运算数进行类型转换，而严格相等与非严格相等运算前不进行类型转换，该功能自然有其存在的意义。

下面看一个严格相等与非严格相等运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-total-equal.html 文件）。

【代码 6-17】

```
01  <script type="text/javascript">
02       var iNum = 123;
03       var iStr = "123";
04       console.log("123 == '123' = " + (iNum == iStr));
05       console.log("123 === '123' = " + (iNum === iStr));
06       console.log("123 != '123' = " + (iNum != iStr));
07       console.log("123 !== '123' = " + (iNum !== iStr));
08       var bR_1_1 = null == undefined;
09       console.log("null == undefined = " + bR_1_1);
10       var bR_1_2 = null === undefined;
11       console.log("null === undefined = " + bR_1_2);
12       var bR_2_1 =  null != undefined;
13       console.log("null != undefined = " + bR_2_1);
14       var bR_2_2 = null !== undefined;
15       console.log("null !== undefined = " + bR_2_2);
16       var bR_NaN_1 = NaN == NaN;
17       console.log("NaN == NaN = " + bR_NaN_1);
18       var bR_NaN_2 = NaN === NaN;
19       console.log("NaN === NaN = " + bR_NaN_2);
20  </script>
```

关于【代码 6-17】的分析如下：

这段代码主要就是对数值、字符串、null、undefined 和 NaN 分别使用严格相等（===）和非严格相等（!==）运算符进行了比较运算。

第 02 和第 03 行代码分别定义了两个变量 iNum 和 iStr，变量 iNum 初始化为数值 123，变量 iStr 初始化为数值型字符串"123"。

第 04～07 行代码分别使用等号（==）、不等号（!=）、严格相等（===）和非严格相等（!==）运算符对变量 iNum 和变量 iStr 进行了比较运算。

第 08～15 行代码分别使用等号（==）、不等号（!=）、严格相等（===）和非严格相等（!==）运算符对原始值 null 和 undefined 进行了比较运算。

第 16～19 行代码分别使用等号（==）和严格相等（===）运算符对 NaN 进行了比较运算。

运行页面，调试信息如图 6.19 所示。

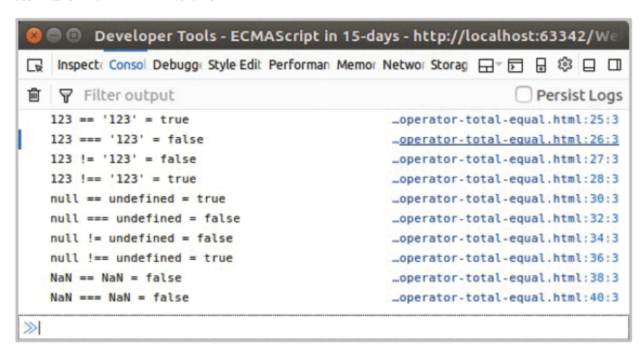


图 6.19　ECMAScript 等性运算符（严格相等与非严格相等）

如图 6.19 所示，从第 04 和第 05 行代码输出的结果来看，使用等号和严格相等运算符对 123 和"123"的比较运算结果是不同的。原因就是前文中提到的，严格相等运算符不会对运算数进行类型转换，自然数字 123 和字符串"123"是不相等的。

从第 06 和第 07 行代码输出的结果来看，使用不等号和非严格相等运算符对 123 和"123"的比较运算结果与使用等号和严格相等运算符的结果正好相反。

第 09 行代码使用等号的输出的结果在【代码 6-16】中已经清楚了，而与之相对应的第 11 行代码使用严格相等输出的结果正好与之相反，即表达式（null === undefined）的返回值为 false。

从第 13 和第 15 行代码输出的结果来看，使用不等号和非严格相等运算符对 null 和 undefined 的比较运算结果与使用等号和严格相等运算符的结果正好相反。

最后，从第 17 和第 19 行代码输出的结果来看，对于原始值 NaN 与其自身，无论是使用等号还是严格相等进行比较运算的结果，返回值均为 false，这一点也是符合 ECMAScript 语法中对原始值 NaN 的定义。

6.6　ECMAScript 位运算符及表达式

在 ECMAScript 语法规范中还定义了位运算符，用于对数字底层（即表示数字的 32 位二进制数）进行操作。具体说到学习位运算符，我们一定要先明白二进制编码的原理，这一点非常重要。另外，如果读者曾阅读过《汇编语言程序设计》或《计算机组成原理》这两本书，学习位运算符自然就不会有什么难度。下面对这些位运算符逐一进行介绍。

6.6.1　位运算符与表达式概述

位运算符用于 32 位二进制数的位操作，且操作结果均转换为十进制的数字。ECMAScript 语法规范中定义的位运算符与表达式详见表 6.6。

表 6.6　ECMAScript 位运算符与表达式

<table>
<tr><th>运算符</th><th>描述</th><th>示例</th><th>二进制表达式</th><th>二进制结果</th><th>十进制结果</th><th>前提条件</th></tr>
<tr><td>&</td><td>与</td><td>x&y</td><td>0101 & 0001</td><td>0001</td><td>1</td><td>x=5, y=1</td></tr>
<tr><td>|</td><td>或</td><td>x|y</td><td>0101 | 0001</td><td>0101</td><td>5</td><td>x=5, y=1</td></tr>
<tr><td>~</td><td>取反</td><td>~x</td><td>~0101</td><td>1010</td><td>10</td><td>x=5</td></tr>
<tr><td>^</td><td>异或</td><td>x^y</td><td>0101 ^ 0001</td><td>0100</td><td>4</td><td>x=5, y=1</td></tr>
<tr><td><<</td><td>左移</td><td>x<<1</td><td>0101 << 1</td><td>1010</td><td>10</td><td>x=5</td></tr>
<tr><td>>></td><td>右移</td><td>x>>1</td><td>0101 >> 1</td><td>0010</td><td>2</td><td>x=5</td></tr>
</table>

6.6.2　整数编码介绍

在计算机操作系统中，一般对整数的编码有两种形式，即有符号整数（允许用正数和负数）和无符号整数（只允许用正数）。而在 ECMAScript 语法规范中，所有整数默认都是有符号的整数。

对于有符号的整数（32 位二进制编码），一般使用前 31 位表示整数的数值，用第 32 位表示整数的符号，具体就是 0 表示正数，1 表示负数。当然，有符号整数的数值范围就从-2147483648～2147483647 之间了，如数字 15 就可以使用图 6.20 来表示。

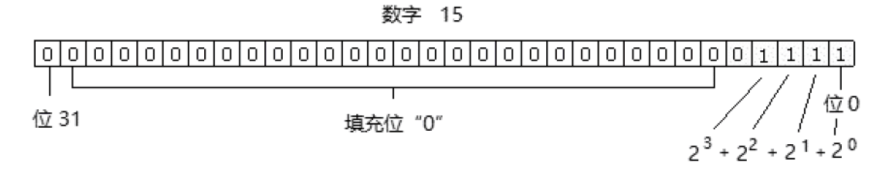


图 6.20　ECMAScript 整数编码表示方法

如图 6.20 所示，前 31 位中的每一位都表示数值 2 的幂。从第 1 位（位 0）开始（表示 2^0），第 2 位（位 1）表示（2^1），第 3 位（位 2）表示（2^2），第 4 位（位 3）表示（2^3），这样一直累加就是（$2^0 + 2^1 + 2^2 + 2^3 = 15$）。而从第 5 位（位 4）开始都没用到的位则用 0 填充。最后，第 32 位（位 31）为 0，表示为正数。

下面看一个应用正整数编码的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-encode-posi-num.html 文件）。

【代码 6-18】

```
01  <script type="text/javascript">
02      var iNum = 15;
03      console.log("15 to binary is " + iNum.toString(2));
04  </script>
```

关于【代码 6-18】的分析如下：

这段代码主要就是将数值 15 转换为二进制数，并进行了输出操作。

运行页面，调试信息如图 6.21 所示。从第 03 行代码输出的结果来看，仅输出了二进制数 1111，而不是全部 32 位二进制数，这是因为填充位“0”被省略掉了。

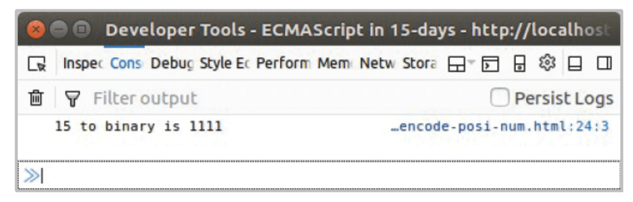


图 6.21　ECMAScript 位运算符（正数编码）

而对于负整数也是存储为二进制代码的，不过采用的形式是二进制补码形式。关于二进制补码的内容，读者可以参考前面提到的《汇编语言程序设计》或《计算机组成原理》这两本书，这里就不深入讨论了。

之所以不对二进制补码进行详细介绍，是因为 ECMAScript 语法规范中对负整数采用了简单的处理方式，即直接使用负号（-）来表示将负整数转换为二进制数的形式。我们还是通过具体的代码实例来了解一下。

下面再看一个应用负整数编码的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-encode-nega-num.html 文件）。

【代码 6-19】

```
01  <script type="text/javascript">
02      var iNum = -15;
03      console.log("-15 to binary is " + iNum.toString(2));
04  </script>
```

关于【代码 6-19】的分析如下：

这段代码主要就是将负数（-15）转换为二进制数，并进行了输出操作。

运行页面，调试信息如图 6.22 所示。

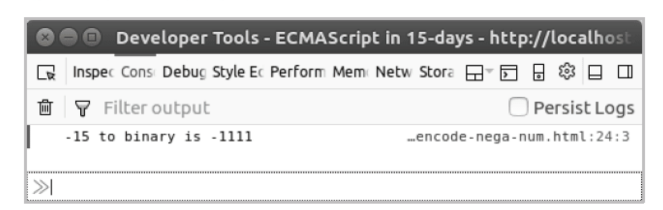


图 6.22　ECMAScript 位运算符（负数编码）

如图 6.22 所示，从第 03 行代码输出的结果来看，负数直接输出了二进制-1111，而不是二进制补码形式，这是因为 ECMAScript 语法规范对其进行了简化处理，也是为了更方便于设计人员设计开发。

最后，再提一下无符号整数的处理方式。因为无符号整数把最后一位（位 31）作为一个具体数位来处理，所以无符号整数的第 32 位不表示数字的符号，而是具体的数值。自然，无符号整数的数值范围就变为从 0～4294967295 了。

注　意

ECMAScript 语法规范中定义所有整数都默认存储为有符号的整数。仅在使用 ECMAScript 的位运算符操作时，才会创建出无符号整数。

6.6.3　NOT 位运算符及表达式

在 ECMAScript 语法规范中，位运算符 NOT 是由否定号（~）来表示的，主要用于二进制算术运算。位运算符 NOT 的操作步骤如下：

第一步，将运算数转换为 32 位二进制数。

第二步，再将第一步得到的二进制数转换为其反码形式。

第三步，最后将二进制数反码转换为浮点数。

关于二进制反码的内容，与二进制补码类似，读者可自行参考前文中提到的书籍。即使不去详细了解二进制反码的内容，也不影响对位运算符 NOT 的使用，因为 ECMAScript 语法规范仍会默认输出十进制数值。

下面看一个 NOT(~)运算符表达式的代码示例(详见源代码 ch06 目录中的 ch06-es-operator-not.html 文件）。

【代码 6-20】

```
01  <script type="text/javascript">
02      var iNum = 255;
03      console.log("255 to binary is " + iNum.toString(2));
04      var iNum_NOT = ~iNum;
05      console.log("NOT 255 to binary is " + iNum_NOT.toString(2));
06      console.log("NOT 255 is " + iNum_NOT.toString());
07  </script>
```

关于【代码 6-20】的分析如下：

这段代码主要就是对数值 255 使用 NOT（~）运算符进行了位操作运算。

第 04 行代码通过 NOT(~)运算符对第 02 行代码中定义的变量 iNum 进行了位操作运算~iNum。

运行页面，调试信息如图 6.23 所示。

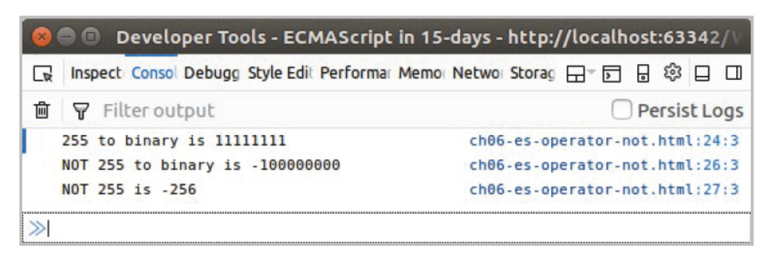


图 6.23　ECMAScript 位运算符（NOT）

如图 6.23 所示，从第 03 行代码输出的结果来看，数值 255 的二进制数形式为 11111111。

从第 05 行代码输出的结果来看，对数值 255 进行 NOT 位运算符操作后，二进制数形式为 -100000000。

从第 06 行代码输出的结果来看，正好是对数值 255 先求反（二进制反码），然后减 1 的结果。

6.6.4　AND 位运算符及表达式

在 ECMAScript 语法规范中，位运算符 AND 是由和号（&）来表示的，主要用于二进制算术运算。使用位运算符 AND 时先要把两个运算数按照位对齐，然后根据表 6.7 中的规则进行运算。

表 6.7　ECMAScript 位运算符（AND）规则

运算数 1	运算数 2	位运算（AND）结果
1	1	1
0	1	0
1	0	0
0	0	0

下面看一个 AND 运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-and.html 文件）。

【代码 6-21】

```
01  <script type="text/javascript">
02      var iNum1 = 255;
03      console.log("255 to binary is " + iNum1.toString(2));
04      var iNum2 = 0xAA;
05      console.log("0xAA to binary is " + iNum2.toString(2));
06      var iNum_AND = iNum1 & iNum2;
07      console.log("255 & 0xAA to binary is " + iNum_AND.toString(2));
08      console.log("255 & 0xAA to hex is " + iNum_AND.toString(16));
09      console.log("255 & 0xAA is " + iNum_AND.toString());
10  </script>
```

关于【代码 6-21】的分析如下：

这段代码主要就是对数值 255 和 0xAA 使用 AND（&）运算符进行了位操作运算。

第 02 行代码中定义了第一个变量 iNum1，并初始化赋值为数值 255。

第 04 行代码中定义了第二个变量 iNum2，并初始化赋值为十六进制数值 0xAA。这里使用十六进制数表示是为了能够更清晰地看出位运算符 AND 的操作结果。

第 06 行代码中通过 AND(&)运算符对变量 iNum1 和变量 iNum2 进行了位操作运算 iNum1 & iNum2，并将结果返回值保存在变量 iNum_AND 中。

第 07～09 行代码中分别对变量 iNum_AND 使用二进制、十六进制和十进制形式进行了输出。

运行页面，调试信息如图 6.24 所示。

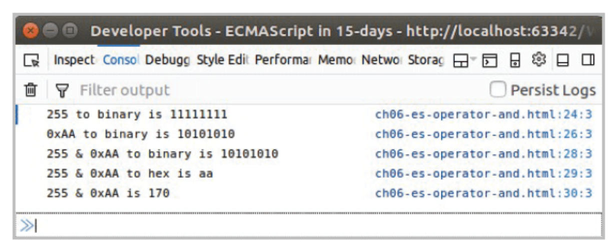


图 6.24　ECMAScript 位运算符（AND）

如图 6.24 所示，从第 03 行代码输出的结果来看，数值 255 的二进制数形式为 11111111。

从第 05 行代码输出的结果来看，十六进制数值 0xAA 的二进制数形式为 10101010。

从第 07 行代码输出的结果来看，数值 255 和 0xAA 通过 AND 位操作运算后，运算结果的二进制形式仍为 10101010，该结果是符合表 6.7 中的运算规则的。

从第 08 行代码输出的结果来看，值 aa 正是十六进制数 0xAA。

从第 09 行代码输出的结果来看，十六进制数 0xAA 对应的十进制数正是数值 170。

6.6.5　OR 位运算符及表达式

在 ECMAScript 语法规范中，位运算符 OR 是由符号（|）来表示的，主要用于二进制算术运算。使用位运算符 OR 时同样要先把两个运算数按照位对齐，然后根据表 6.8 中的规则进行运算。

表 6.8　ECMAScript 位运算符（OR）规则

运算数 1	运算数 2	位运算（OR）结果
1	1	1
0	1	1
1	0	1
0	0	0

下面看一个 OR 运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-or.html 文件），这段代码是在【代码 6-21】的基础上稍加修改而成的，读者可以进行参考对比。

【代码 6-22】

```
01  <script type="text/javascript">
02      var iNum1 = 255;
03      console.log("255 to binary is " + iNum1.toString(2));
04      var iNum2 = 0xAA;
05      console.log("0xAA to binary is " + iNum2.toString(2));
06      var iNum_OR = iNum1 | iNum2;
07      console.log("255 | 0xAA to binary is " + iNum_OR.toString(2));
08      console.log("255 | 0xAA to hex is " + iNum_OR.toString(16));
09      console.log("255 | 0xAA is " + iNum_OR.toString());
10  </script>
```

关于【代码 6-22】的分析如下：

这段代码主要就是对数值 255 和 0xAA 使用 OR 运算符进行了位操作运算。

第 02 行代码中定义了第一个变量 iNum1，并初始化赋值为数值 255。

第 04 行代码中定义了第二个变量 iNum2，并初始化赋值为十六进制数值 0xAA。这里使用十六进制数表示是为了能够更清晰地看出位运算符 OR 的操作结果。

第 06 行代码中通过 OR（|）运算符对变量 iNum1 和变量 iNum2 进行了位操作运算 iNum1 | iNum2，并将结果返回值保存在变量 iNum_OR 中。

第 07～09 行代码中分别对变量 iNum_OR 使用二进制、十六进制和十进制形式进行了输出。

运行页面，调试信息如图 6.25 所示。

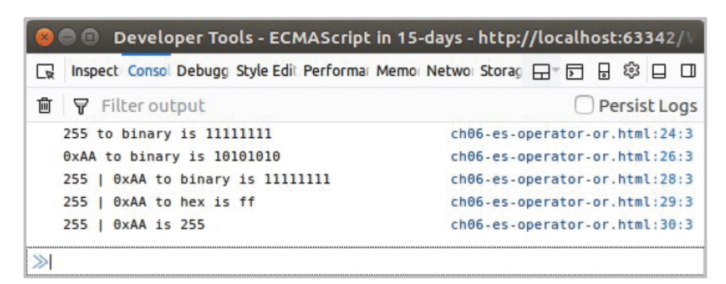


图 6.25　ECMAScript 位运算符（OR）

如图 6.25 所示，从第 03 行代码输出的结果来看，数值 255 的二进制数形式为 11111111。

从第 05 行代码输出的结果来看，十六进制数值 0xAA 的二进制数形式为 10101010。

从第 07 行代码输出的结果来看，数值 255 和 0xAA 通过 OR 位操作运算后，运算结果的二进制形式仍为 11111111，该结果是符合表 6.8 中的运算规则的。

从第 08 行代码输出的结果来看，值 ff 正是十六进制数 0xAA。

从第 09 行代码输出的结果来看，十六进制数 0xAA 对应的十进制数正是数值 255。

6.6.6　XOR 位运算符及表达式

在 ECMAScript 语法规范中，位运算符 XOR 是由符号（^）来表示的，主要用于二进制算术运算。使用位运算符 XOR 时同样要把先两个运算数按照位对齐，然后根据表 6.9 中的规则进行运算。

表 6.9　ECMAScript 位运算符（XOR）规则

运算数 1	运算数 2	位运算（XOR）结果
1	1	0
0	1	1
1	0	1
0	0	0

下面看一个 XOR 运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-xor.html 文件），这段代码是在【代码 6-21】和【代码 6-22】的基础上稍加修改而成的，读者可以进行参考对比。

【代码 6-23】

```
01  <script type="text/javascript">
02      var iNum1 = 255;
03      console.log("255 to binary is " + iNum1.toString(2));
04      var iNum2 = 0xAA;
05      console.log("0xAA to binary is " + iNum2.toString(2));
06      var iNum_XOR = iNum1 ^ iNum2;
07      console.log("255 ^ 0xAA to binary is " + iNum_XOR.toString(2));
08      console.log("255 ^ 0xAA to hex is " + iNum_XOR.toString(16));
09      console.log("255 ^ 0xAA is " + iNum_XOR.toString());
10  </script>
```

关于【代码 6-23】的分析如下：

这段代码主要就是对数值 255 和 0xAA 使用 XOR 运算符进行了位操作运算。

第 02 行代码中定义了第一个变量 iNum1，并初始化赋值为数值 255。

第 04 行代码中定义了第二个变量 iNum2，并初始化赋值为十六进制数值 0xAA。这里使用十六进制数表示是为了能够更清晰地看出位运算符 XOR 的操作结果。

第 06 行代码中通过 XOR（^）运算符对变量 iNum1 和变量 iNum2 进行了位操作运算 iNum1 ^ iNum2，并将结果返回值保存在变量 iNum_XOR 中。

第 07～09 行代码中分别对变量 iNum_XOR 使用二进制、十六进制和十进制形式进行了输出。

运行页面，调试信息如图 6.26 所示。

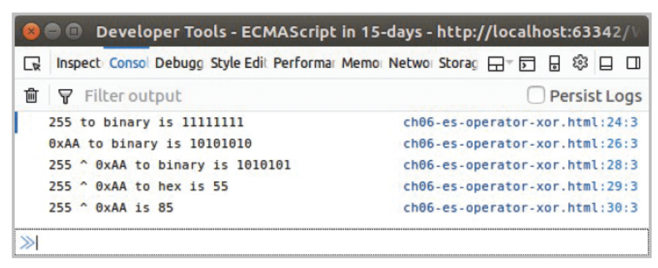


图 6.26　ECMAScript 位运算符（XOR）

如图 6.26 所示，从第 03 行代码输出的结果来看，数值 255 的二进制数形式为 11111111。

从第 05 行代码输出的结果来看，十六进制数值 0xAA 的二进制数形式为 10101010。

从第 07 行代码输出的结果来看，数值 255 和 0xAA 通过 XOR 位操作运算后，运算结果的二进制形式为 1010101，写成 8 位二进制就是 01010101，该结果是符合表 6.9 中的运算规则的。

从第 08 行代码输出的结果来看，值 55 正是十六进制数 0xAA。

从第 09 行代码输出的结果来看，十六进制数 0xAA 对应的十进制数正是数值 85。

6.6.7　左移运算符及表达式

在 ECMAScript 语法规范中，左移运算符是由符号（<<）来表示的，主要用于实现将数字中的所有数位向左移动指定数量位数的二进制算术运算。

对于在左移运算时数字右边可能会多出的若干空位，将使用数字 0 来填充，保证结果成为完整的 32 位二进制数。

需要注意的是，左移运算会保留数字的符号位（第 32 位）。不过设计人员不用担心，一般不能直接访问第 32 位符号位，一切都是通过 ECMAScript 语法规范在后台实现的。

下面看一个左移（<<）运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-shl.html 文件）。

【代码 6-24】

```
01  <script type="text/javascript">
02      var iNum = 255;
03      console.log("255 to binary is " + iNum.toString(2));
04      var iNum_shl_2 = iNum << 2;
05      console.log("255 << 2 to binary is " + iNum_shl_2.toString(2));
06      console.log("255 << 2  is " + iNum_shl_2.toString());
```

```
07        var iNum_shl_8 = iNum << 8;
08        console.log("255 << 8 to binary is " + iNum_shl_8.toString(2));
09        console.log("255 << 8  is " + iNum_shl_8.toString());
10        var iNum_shl_16 = iNum << 16;
11        console.log("255 << 16 to binary is " + iNum_shl_16.toString(2));
12        console.log("255 << 16  is " + iNum_shl_16.toString());
13        var iNum_shl_32 = iNum << 32;
14        console.log("255 << 32 to binary is " + iNum_shl_32.toString(2));
15        console.log("255 << 32  is " + iNum_shl_32.toString());
16    </script>
```

关于【代码 6-24】的分析如下：

这段代码主要就是对数值 255 使用左移（<<）运算符进行了位操作运算。

第 02 行代码中定义了一个变量 iNum，并初始化赋值为数值 255。

第 04 行代码中使用左移（<<）运算符对变量 iNum 进行了两位左移运算操作（iNum << 2）。

第 07 行代码中使用左移（<<）运算符对变量 iNum 进行了 8 位左移运算操作（iNum << 8）。

第 10 行代码中使用左移（<<）运算符对变量 iNum 进行了 16 位左移运算操作（iNum << 16）。

第 13 行代码中使用左移（<<）运算符对变量 iNum 进行了 32 位左移运算操作（iNum << 32）。

运行页面，调试信息如图 6.27 所示。

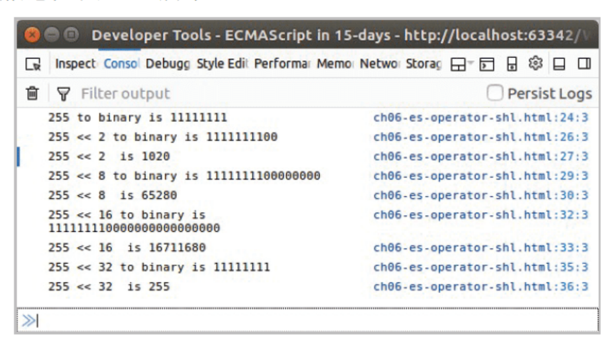


图 6.27　ECMAScript 位运算符（左移）

如图 6.27 所示，从第 03 行代码输出的结果来看，数值 255 的二进制数形式为 11111111。

从第 05 和第 06 行代码输出的结果来看，对数值 255 左移两位后的结果为 1111111100，右边空位自动补 0 了。

从第 08 和第 09 行代码输出的结果来看，对数值 255 左移 8 位后的结果为 1111111100000000，右边空位同样自动补 0 了。

从第 11 和第 12 行代码输出的结果来看，对数值 255 左移 16 位后的结果为 111111110000000000000000，右边空位依然自动补 0 了。

从第 14 和第 15 行代码输出的结果来看，对数值 255 左移 32 位后的结果为 11111111，说明左移 32 位的操作结果仍为初始值。

6.6.8　保留符号位的右移运算符及表达式

在 ECMAScript 语法规范中，保留符号位的右移运算符是由符号（>>）来表示的，主要用于实现

将数字中的所有数位向右移动指定数量位数，且保留该数的符号（正号或负号）位的二进制算术运算。

对于在右移运算时数字左边可能会多出的若干空位，将使用数字 0 来填充，保证结果成为完整的 32 位二进制数。

下面看一个保留符号位的右移（>>）运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-shr.html 文件）。

【代码 6-25】

```
01  <script type="text/javascript">
02      var iNum = 2139095040;
03      console.log("2139095040 to binary is " + iNum.toString(2));
04      var iNum_shr_2 = iNum >> 2;
05      console.log("2139095040 >> 2 to binary is " + iNum_shr_2.toString(2));
06      console.log("2139095040 >> 2  is " + iNum_shr_2.toString());
07      var iNum_shr_8 = iNum >> 8;
08      console.log("2139095040 >> 8 to binary is " + iNum_shr_8.toString(2));
09      console.log("2139095040 >> 8  is " + iNum_shr_8.toString());
10      var iNum_shr_16 = iNum >> 16;
11      console.log("2139095040 >> 16 to binary is " + iNum_shr_16.toString(2));
12      console.log("2139095040 >> 16  is " + iNum_shr_16.toString());
13      var iNum_shr_32 = iNum << 32;
14      console.log("2139095040 >> 32 to binary is " + iNum_shr_32.toString(2));
15      console.log("2139095040 >> 32  is " + iNum_shr_32.toString());
16  </script>
```

关于【代码 6-25】的分析如下：

这段代码主要就是对数值 2139095040 使用右移（>>）运算符进行了保留符号位的位操作运算。

第 02 行代码中定义了一个变量 iNum，并初始化赋值为数值 2139095040。

第 04 行代码中使用右移（>>）运算符对变量 iNum 进行了两位右移运算操作（iNum >> 2）。

第 07 行代码中使用右移（>>）运算符对变量 iNum 进行了 8 位右移运算操作（iNum >> 8）。

第 10 行代码中使用右移（>>）运算符对变量 iNum 进行了 16 位右移运算操作（iNum >> 16）。

第 13 行代码中使用右移（>>）运算符对变量 iNum 进行了 32 位右移运算操作（iNum >> 32）。

运行页面，调试信息如图 6.28 所示。

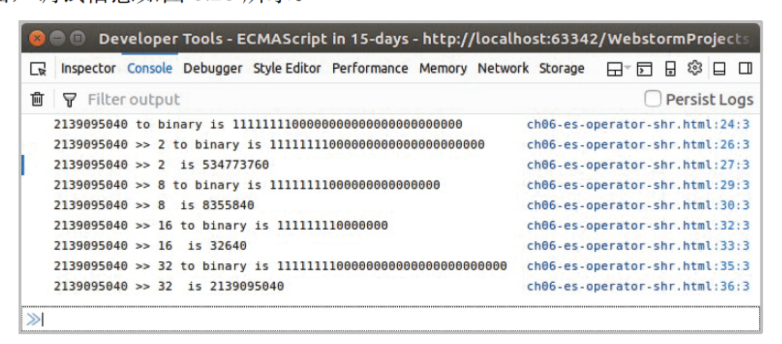


图 6.28　ECMAScript 位运算符（保留符号位的右移）

如图 6.28 所示，从第 03 行代码输出的结果来看，数值 2139095040 的二进制数形式为 1111111100000000000000000000000。

从第 05 和第 06 行代码输出的结果来看，对数值 2139095040 右移两位后的结果为

111111110000000000000000000000。

从第 08 和第 09 行代码输出的结果来看，对数值 2139095040 右移 8 位后的结果为 1111111100000000000000000。

从第 11 和第 12 行代码输出的结果来看，对数值 2139095040 右移 16 位后的结果为 111111110000000。

从第 14 和第 15 行代码输出的结果来看，对数值 2139095040 右移 32 位后的结果仍为 2139095040，说明保留符号位的右移 32 位的操作结果仍为初始值。

6.6.9 无符号位的右移运算符及表达式

在 ECMAScript 语法规范中，无符号位的右移运算符是由符号（>>>）来表示的，主要用于实现将数字中的所有数位（包括第 32 位的符号位）整体向右移动指定数量位数的二进制算术运算。

对于正数的无符号位右移运算，其结果与保留符号位的右移运算是一致的。但对于负数来说，无符号位的右移运算与保留符号位的右移运算的结果就完全不一致了。

下面看一个无符号位的右移（>>>）运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-shr-nosign.html 文件）。

【代码 6-26】

```
01  <script type="text/javascript">
02      var iNum = 256;
03      console.log("256 to binary is " + iNum.toString(2));
04      var iNum_shr = iNum >> 8;
05      console.log("256 >> 8 to binary is " + iNum_shr.toString(2));
06      console.log("256 >> 8 is " + iNum_shr.toString());
07      var iNum_shr_nosign = iNum >>> 8;
08      console.log("256 >>> 8 to binary is " + iNum_shr_nosign.toString(2));
09      console.log("256 >>> 8 is " + iNum_shr_nosign.toString());
10      var iNum_minus = -256;
11      console.log("-256 to binary is " + iNum_minus.toString(2));
12      var iNum_shr_minus = iNum_minus >> 8;
13      console.log("-256 >> 8 to binary is " + iNum_shr_minus.toString(2));
14      console.log("-256 >> 8 is " + iNum_shr_minus.toString());
15      var iNum_shr_nosign_minus = iNum_minus >>> 8;
16      console.log("-256 >>> 8 to binary is " + iNum_shr_nosign_minus.toString(2));
17      console.log("-256 >>> 8 is " + iNum_shr_nosign_minus.toString());
18  </script>
```

关于【代码 6-26】的分析如下：

这段代码主要就是对正数 256 和负数-256，同时分别使用保留符号位的右移（>>）运算符和无符号位的右移（>>>）运算符进行了位操作运算，目的就是验证一下无符号位的右移（>>>）运算符对于正数和负数的不同操作结果。

运行页面，调试信息如图 6.29 所示。

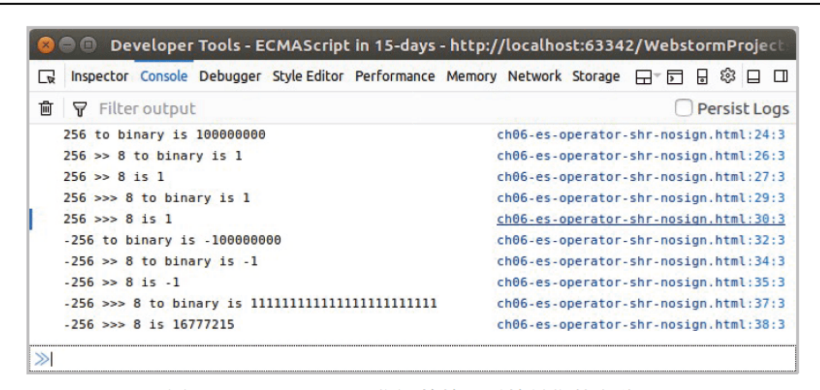


图 6.29　ECMAScript 位运算符（无符号位的右移）

如图 6.29 所示，从第 03 行代码输出的结果来看，数值 256 的二进制数形式为 100000000。

从第 04～06 行代码输出的结果来看，对数值 256 使用保留符号位的右移（>>）运算符右移 8 位后的结果为 1。

从第 07～09 行代码输出的结果来看，对数值 256 使用无符号位的右移（>>>）运算符右移 8 位后的结果仍为 1。

这就说明对于正数，保留符号位的右移（>>）运算符和无符号位的右移（>>>）运算符操作的结果是一致的。

从第 11 行代码输出的结果来看，数值-256 的二进制数形式为-100000000。

从第 12～14 行代码输出的结果来看，对数值-256 使用保留符号位的右移（>>）运算符右移 8 位后的结果为-1。

而从第 15～17 行代码输出的结果来看，对数值-256 使用无符号位的右移（>>>）运算符右移 8 位后的结果仍为 16777215。

这就说明对于负数，保留符号位的右移（>>）运算符和无符号位的右移（>>>）运算符操作的结果是不一致的。因为无符号位的右移运算会将符号位的数值 1 一起右移，所以操作后数值结果会发生变化。

6.7　ECMAScript 逻辑运算符及表达式

在 ECMAScript 语法规范中，逻辑运算占有非常重要的地位。为什么这么说呢？因为在程序中会有大量的条件判断语句是依赖于逻辑运算来完成的。

6.7.1　逻辑运算符与表达式概述

逻辑运算符用来确定变量或值之间的逻辑关系。ECMAScript 语法规范中定义的逻辑运算符与表达式详见表 6.10。

表 6.10　ECMAScript 逻辑运算符与表达式

运算符	描述	示例	结果	前提条件
&&	与（AND）	(x<2)&&(y>0)	true	x=1, y=1
||	或（OR）	(x<1)||(y>1)	true	x=0, y=2
!	非（NOT）	!(x==y)	false	x=1, y=1

6.7.2　ToBoolean 逻辑值转换操作

在 ECMAScript 语法规范中，定义了转换逻辑值的 ToBoolean 操作，用于将各种类型的值转换为逻辑值，具体规则详见表 6.11。

表 6.11　ToBoolean 规则

参数类型	ToBoolean 操作结果
Null	false
Undefined	false
Number	如果参数为+0,-0 或 NaN，结果为 false；否则为 true
String	如果参数为空字符串，则结果为 false；否则为 true
Boolean	结果等于输入的参数（不转换）
Object	true

下面看一个 ToBoolean 操作的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-toBoolean.html 文件）。

【代码 6-27】

```
01  <script type="text/javascript">
02      console.log("ToBoolean(null) = " + Boolean(null));
03      console.log("ToBoolean(Undefined) = " + Boolean(undefined));
04      console.log("ToBoolean(true) = " + Boolean(true));
05      console.log("ToBoolean(false) = " + Boolean(false));
06      console.log("ToBoolean(+0) = " + Boolean(+0));
07      console.log("ToBoolean(-0) = " + Boolean(-0));
08      console.log("ToBoolean(NaN) = " + Boolean(NaN));
09      console.log("ToBoolean(1) = " + Boolean(1));
10      console.log("ToBoolean('ECMAScript') = " + Boolean("ECMAScript"));
11      console.log("ToBoolean('') = " + Boolean(""));
12  </script>
```

关于【代码 6-27】的分析如下：

这段代码主要就是使用 ToBoolean 操作，将一些原始值或特殊值转换为 Boolean 类型的值。这里需要注意的是，在 ECMAScript 语法规范中并没有 ToBoolean 这个方法（这与 String 类型的 toString()方法是不同的），不过可以使用 Boolean()方法进行强制类型转换。

运行页面，调试信息如图 6.30 所示。

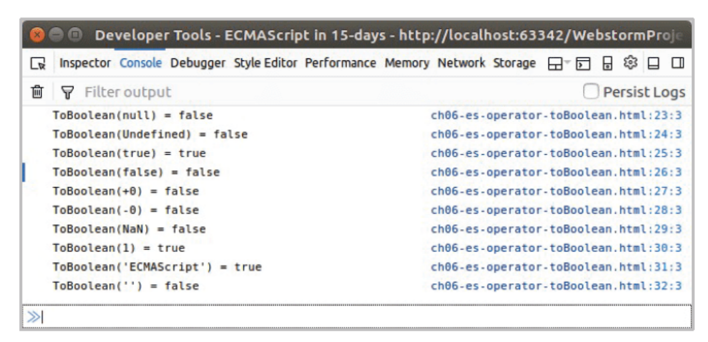


图 6.30　ECMAScript 逻辑运算（ToBoolean 操作）

如图 6.30 所示，从第 02 和第 03 行代码输出的结果来看，通过 Boolean()方法对原始值 null 和 undefined 强制类型转换后，返回值均是布尔值 false。

从第 04 和第 05 行代码输出的结果来看，通过 Boolean()方法对布尔值强制类型转换后，返回值仍是原布尔值。

从第 06 和第 07 行代码输出的结果来看，通过 Boolean()方法对+0 和-0 强制类型转换后，返回值均是布尔值 false。

从第 08 行代码输出的结果来看，通过 Boolean()方法对原始值 NaN 强制类型转换后，返回值是布尔值 false。

从第 09 行代码输出的结果来看，通过 Boolean()方法对数值 1 强制类型转换后，返回值是布尔值 true。

从第 10 和第 11 行代码输出的结果来看，通过 Boolean()方法对字符串强制类型转换后，非空字符串返回值是布尔值 true，空字符串返回值是 false。

6.7.3　AND 运算符及表达式

在 ECMAScript 语法规范中，AND 运算符用于执行逻辑“与”运算，由双和号（&&）来表示。ECMAScript 语法规范中定义的逻辑 AND 运算符的规则详见表 6.12。

表 6.12　ECMAScript 逻辑运算符（AND）规则

运算数 1	运算数 2	逻辑运算（AND）结果
true	true	true
true	false	false
false	true	false
false	false	false

另外，对于逻辑 AND 运算中的运算数可以是任何类型，不一定非是 Boolean 类型值。如果某个运算数不是原始的 Boolean 型值，那么逻辑 AND 运算后的结果不一定返回 Boolean 类型值。

ECMAScript 语法规范中对这方面的内容具体说明如下：

- 如果一个运算数是对象，另一个是 Boolean 类型值 true，则会返回该对象。
- 如果两个运算数都是对象，则返回第二个对象。

- 如果某个运算数是原始值 null，则返回原始值 null。
- 如果某个运算数是原始值 NaN，则返回原始值 NaN。
- 如果某个运算数是原始值 undefined，则返回原始值 undefined。

下面看一个逻辑 AND 运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-logical-and.html 文件）。

【代码 6-28】

```
01  <script type="text/javascript">
02      var b_11 = true && true;
03      console.log("true && true = " + b_11);
04      var b_10 = true && false;
05      console.log("true && false = " + b_10);
06      var b_01 = false && true;
07      console.log("false && true = " + b_01);
08      var b_00 = false && false;
09      console.log("false && false = " + b_00);
10      var b_null = null && true;
11      console.log("null && true = " + b_null);
12      var b_NaN = NaN && true;
13      console.log("NaN && true = " + b_NaN);
14      var b_undefined = undefined && true;
15      console.log("undefined && true = " + b_undefined);
16  </script>
```

关于【代码 6-28】的分析如下：

这段代码主要就是使用逻辑 AND 运算符（&&）对原始值和特殊值进行了逻辑“与”操作运算。

运行页面，调试信息如图 6.31 所示。

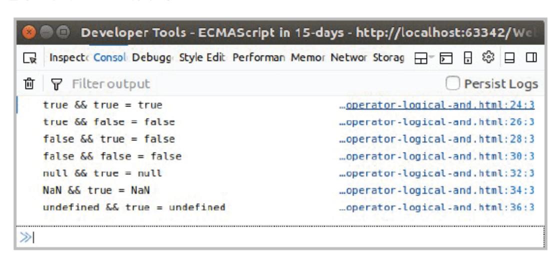


图 6.31　ECMAScript 逻辑运算符（AND）

如图 6.31 所示，从第 03、第 05、第 07 和第 09 行代码输出的结果来看，Boolean 值之间的逻辑 AND 运算返回值与表 6.12 中的定义是一致的。

从第 11、第 13 和第 15 行代码输出的结果来看，对于原始值 null、NaN 和 undefined 与其他运算数之间的逻辑 AND 运算返回值仍是其本身。

6.7.4　OR 运算符及表达式

在 ECMAScript 语法规范中，OR 运算符用于执行逻辑“或”运算，由符号(||)来表示。ECMAScript 语法规范中定义的逻辑 OR 运算符的规则详见表 6.13。

表 6.13　ECMAScript 逻辑运算符（OR）规则

运算数 1	运算数 2	逻辑运算（OR）结果
true	true	true
true	false	true
false	true	true
false	false	false

另外，与逻辑 AND 运算符一样，逻辑 OR 运算中的运算数可以是任何类型，不一定非是 Boolean 类型值。如果某个运算数不是原始的 Boolean 型值，那么逻辑 OR 运算后的结果不一定返回 Boolean 类型值。

ECMAScript 语法规范中对 OR 运算的具体说明如下：

- 如果一个运算数是对象且其左边的运算数值均为 false，则返回该对象。
- 如果两个运算数都是对象，则返回第一个对象。
- 如果最后一个运算数是 null，并且其他运算数值均为 false，则返回 null。
- 如果最后一个运算数是 NaN，并且其他运算数值均为 false，则返回 NaN。
- 如果最后一个运算数是 undefined，并且其他运算数值均为 false，则返回 undefined。

下面看一个逻辑 OR 运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-logical-or.html 文件）。

【代码 6-29】

```
01  <script type="text/javascript">
02      var b_11 = true || true;
03      console.log("true || true = " + b_11);
04      var b_10 = true || false;
05      console.log("true || false = " + b_10);
06      var b_01 = false || true;
07      console.log("false || true = " + b_01);
08      var b_00 = false || false;
09      console.log("false || false = " + b_00);
10      var b_null_true = null || true;
11      console.log("null || true = " + b_null_true);
12      var b_true_null = true || null;
13      console.log("true || null = " + b_true_null);
14      var b_null_false = null || false;
15      console.log("null || false = " + b_null_false);
16      var b_false_null = false || null;
17      console.log("false || null = " + b_false_null);
18      var b_NaN_true = NaN || true;
19      console.log("NaN || true = " + b_NaN_true);
20      var b_true_NaN = true || NaN;
21      console.log("true || NaN = " + b_true_NaN);
22      var b_NaN_false = NaN || false;
23      console.log("NaN || false = " + b_NaN_false);
24      var b_false_NaN = false || NaN;
25      console.log("false || NaN = " + b_false_NaN);
26      var b_undefined_true = undefined || true;
27      console.log("undefined || true = " + b_undefined_true);
```

```
28        var b_true_undefined = true || undefined;
29        console.log("true || undefined = " + b_true_undefined);
30        var b_undefined_false = undefined || false;
31        console.log("undefined || false = " + b_undefined_false);
32        var b_false_undefined = false || undefined;
33        console.log("false || undefined = " + b_false_undefined);
34    </script>
```

关于【代码 6-29】的分析如下：

这段代码主要就是使用逻辑 OR（||）运算符对原始值和特殊值进行了逻辑“或”操作运算。

运行页面，调试信息如图 6.32 所示。



图 6.32　ECMAScript 逻辑运算符（OR）

如图 6.32 所示，从第 03、第 05、第 07 和第 09 行代码输出的结果来看，Boolean 值之间的逻辑 OR 运算返回值与表 6.13 中的定义是一致的。

从第 11、第 13、第 15 和第 17 行代码输出的结果来看，原始值 null 与 Boolean 值的逻辑 OR 运算返回值比较复杂。null 与 true 逻辑 OR 运算，不论前后顺序如何均会返回 true。而 null 与 false 逻辑 OR 运算，若 null 不在表达式最后，则返回 false；若 null 在表达式最后，则返回 null，这与前文中的描述是一致的。

同样地，原始值 NaN 和 undefined 与 Boolean 值的逻辑 OR 运算返回值，与原始值 null 是类似的。

6.7.5　NOT 运算符及表达式

在 ECMAScript 语法规范中，NOT 运算符用于执行逻辑“非”运算，由感叹号（!）来表示。NOT 运算符与逻辑 AND 运算符和逻辑 OR 运算符不同的是，逻辑 NOT 运算符的返回值一定是 Boolean 类型值。

ECMAScript 语法规范中对逻辑 NOT 运算符内容的具体说明如下：

- 如果运算数是对象，则返回值为 false。
- 如果运算数是数字 0，则返回值为 true。
- 如果运算数是 0 以外的任何数字，则返回值为 false。

- 如果运算数是 null，则返回值为 true。
- 如果运算数是 NaN，则返回值为 true。
- 如果运算数是 undefined，则返回值为 true。

下面看一个逻辑 NOT 运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-logical-not.html 文件）。

【代码 6-30】

```
01  <script type="text/javascript">
02      var b_01 = !true;
03      console.log("!true = " + b_01);
04      var b_00 = !false;
05      console.log("!false = " + b_00);
06      var b_0 = !0;
07      console.log("!0 = " + b_0);
08      var b_1 = !1;
09      console.log("!1 = " + b_1);
10      var b_null = !null;
11      console.log("!null = " + b_null);
12      var b_NaN = !NaN;
13      console.log("!NaN = " + b_NaN);
14      var b_undefined = !undefined;
15      console.log("!undefined = " + b_undefined);
16      var obj = new Object();
17      console.log("!object = " + !obj);
18  </script>
```

关于【代码 6-30】的分析如下：

这段代码主要就是使用逻辑 NOT（!）运算符对原始值和特殊值进行了逻辑“非”操作运算。运行页面，调试信息如图 6.33 所示。

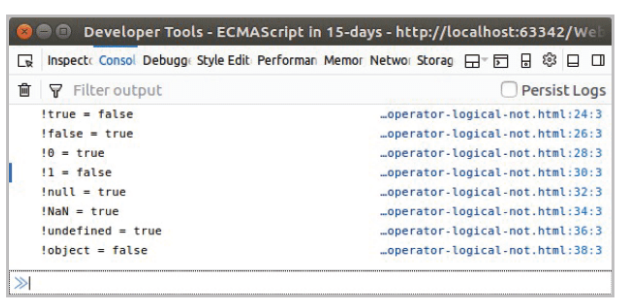


图 6.33　ECMAScript 逻辑运算符（NOT）

如图 6.33 所示，从第 03、第 05、第 07 和第 09 行代码输出的结果来看，对 Boolean 值、0 和 1 的逻辑 NOT 运算返回值与预期是一致的。

从第 11、第 13 和第 15 行代码输出的结果来看，对原始值 null、NaN 和 undefined 的逻辑 NOT 运算返回值均为 true。

从第 17 行代码输出的结果来看，对象类型的逻辑 NOT 运算返回值为 false。

以上就是对逻辑 NOT（!）运算符的测试，读者可以对比前面介绍的逻辑 AND（&&）运算符和逻辑 OR（||）运算符来进行学习。

6.8 ECMAScript 赋值运算符及表达式

在 ECMAScript 语法规范中，赋值运算符是由等号（=）来表示的，用于把等号右边的值赋给等号左边的变量。当然，在具体编程实践中还可以将加性、乘性或位运算符与赋值运算符组合起来使用，以提高代码的简洁性，称为复合赋值运算符。

ECMAScript 语法规范中定义的赋值运算符的规则见表 6.14。

表 6.14　ECMAScript 赋值运算符规则

运算符	示例	等价于	结果	前提条件
=	x=y		x=1	y=1
+=	x+=y	x=x+y	x=2	x=1, y=1
-=	x-=y	x=x-y	x=0	x=1, y=1
*=	x*=y	x=x*y	x=2	x=1, y=2
/=	x/=y	x=x/y	x=2	x=2, y=1
%=	x%=y	x=x%y	x=1	x=3, y=2
<<=	x<<=y	x=x<<y	x=2	x=1,y=1
>>=	x>>=y	x=x>>y	x=0	x=1,y=1
>>>=	x>>>=y	x=x>>>y	x=2147483647	x=-1,y=1

下面看一个使用赋值运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-assign.html 文件）。

【代码 6-31】

```
01  <script type="text/javascript">
02      var i = 1;
03      console.log("i = " + i);
04      var j = i;
05      console.log("j = i is " + j);
06      j += i;
07      console.log("j += i is " + j);
08      j *= j;
09      console.log("j *= j is " + j);
10      j -= i;
11      console.log("j -= i is " + j);
12      j /= i;
13      console.log("j /= i is " + j);
14      j %= 2;
15      console.log("j %= 2 is " + j);
16      j <<= j;
17      console.log("j <<= j is " + j);
18      j >>= j;
19      console.log("j >>= j is " + j);
20      var n = -1;
21      console.log("n = " + n);
22      n >>>= 1;
```

```
23        console.log("n >>>= 1 is " + n);
24    </script>
```

关于【代码 6-31】的分析如下：

这段代码主要就是对数值使用各种赋值运算符进行运算，具体包括加法赋值（+=）、减法赋值（-=）、乘法赋值（*=）、除法赋值（/=）、取模赋值（%=）、左移赋值（<<=）、有符号右移/赋值（>>=）、无符号右移/赋值（>>>=）等几种复合赋值运算的过程及结果。

运行页面，调试信息如图 6.34 所示。

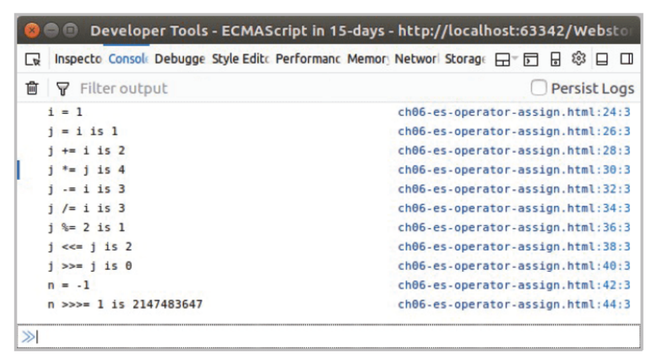


图 6.34　ECMAScript 赋值运算符

如图 6.34 所示，第 02 行代码定义了变量 i，并初始化赋值为 1。

第 04 行代码定义了变量 j，并初始化赋值为变量 i 的值。

从第 07 行代码输出的结果来看，变量 j 进行加法赋值 j += i 后，返回值为 2。

从第 09 行代码输出的结果来看，变量 j 进行乘法赋值 j *= j 后，返回值为 4。

从第 11 行代码输出的结果来看，变量 j 进行减法赋值 j -= i 后，返回值为 3。

从第 13 行代码输出的结果来看，变量 j 进行除法赋值 j /= i 后，返回值仍为 3。

从第 15 行代码输出的结果来看，变量 j 进行取模赋值 j %= 2 后，返回值为 1。

从第 17 行代码输出的结果来看，变量 j 进行左移赋值 j <<= j 后，返回值为 2。

从第 19 行代码输出的结果来看，变量 j 进行保留符号位的右移赋值 j >>= j 后，返回值为 0。

第 20 行代码定义了变量 n，并初始化赋值为负整数-1。

第 21 行代码对变量 n 进行无符号位的右移赋值 j >>>= 1。

从第 22 行代码输出的结果来看，无符号位的右移赋值>>>=操作返回值为 2147483647。

6.9　ECMAScript 条件运算符及表达式

在 ECMAScript 语法规范中，条件运算符是表现形式上相对较复杂的一种，实际上将其理解为一种特殊的表达式似乎更为恰当。ECMAScript 语法规范中对于条件运算符表达式的具体定义如下：

```
variable = boolean_expression ? true_value : false_value;
```

如何理解上面这段关于条件运算符表达式的定义呢？首先，要判断布尔表达式

boolean_expression 的取值，若为 true，则将 true_value 赋值给变量 variable，若为 false，则将 false_value 赋值给变量 variable。同时，对于条件运算符表达式的格式要严格按照上面的书写方式。

下面看一个使用条件运算符表达式的代码示例（详见源代码 ch06 目录中的 ch06-es-operator-conditional.html 文件）。

【代码 6-32】

```
01  <script type="text/javascript">
02      var es = "ECMAScript";
03      var js = "javascript";
04      var vReturn_greater = (es > js) ? "ECMAScript" : "javascript";
05      console.log('(es > js) ? "ECMAScript" : "javascript" = ' + vReturn_greater);
06      var vReturn_less = (es < js) ? "ECMAScript" : "javascript";
07      console.log('(es < js) ? "ECMAScript" : "javascript" = ' + vReturn_less);
08      var vReturn_true = (true) ? "ECMAScript" : "javascript";
09      console.log('(true) ? "ECMAScript" : "javascript" = ' + vReturn_true);
10      var vReturn_false = (false) ? "ECMAScript" : "javascript";
11      console.log('(false) ? "ECMAScript" : "javascript" = ' + vReturn_false);
12      var vReturn_1 = (1) ? "ECMAScript" : "javascript";
13      console.log('(1) ? "ECMAScript" : "javascript" = ' + vReturn_1);
14      var vReturn_0 = (0) ? "ECMAScript" : "javascript";
15      console.log('(0) ? "ECMAScript" : "javascript" = ' + vReturn_0);
16  </script>
```

关于【代码 6-32】的分析如下：

第 02 和第 03 行代码定义了两个字符串变量 es 和 js，并分别初始化赋值为"ECMAScript"和"javascript"。

第 04 行代码使用条件运算符判断布尔表达式 es > js 的结果，为“真”则返回字符串"ECMAScript"，否则返回字符串"javascript"。

同样，第 06 行代码使用条件运算符判断布尔表达式 es < js 的结果，为“真”则返回字符串"ECMAScript"，否则返回字符串"javascript"。

而第 08 和第 10 行代码中，条件运算符的布尔表达式直接为 true 和 false，相当于直接确认返回第一个或第二个返回值。

同样，第 12 和第 14 行代码中，条件运算符的布尔表达式直接为 1 和 0，等同于 true 和 false。

运行页面，调试信息如图 6.35 所示。

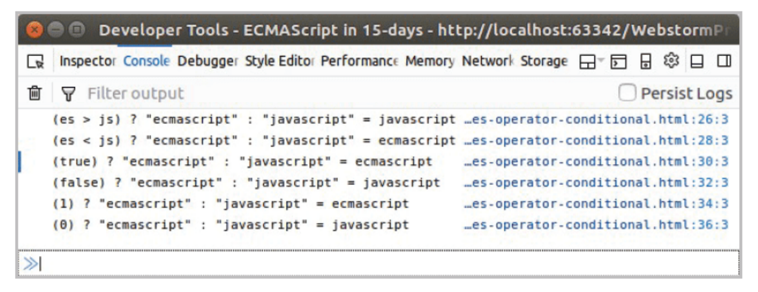


图 6.35　ECMAScript 条件运算符

如图 6.35 所示，从第 05 和第 07 行代码输出的结果来看，字符串进行比较的布尔表达式如果为 true，则返回第一个返回值"ECMAScript"，否则返回第二个返回值"javascript"。

从第 09 和第 11 行代码输出的结果来看，如过布尔表达式直接定义为 true，则返回第一个返回值"ECMAScript"；如果直接定义为 false，则返回第二个返回值"javascript"。

同样，从第 13 和第 15 行代码输出的结果来看，如果布尔表达式直接定义为数值 1，则返回第一个返回值"ECMAScript"；如果直接定义为数值 0，则返回第二个返回值"javascript"。

6.10　本章小结

本章主要介绍了 ECMAScript 语法规范中关于运算符和表达式的内容，具体包括加性运算符、乘性运算符、一元运算符、关系运算符、等性运算符、位运算符、逻辑运算符、赋值运算符、条件运算符及表达式等方面的内容，并通过一些具体实例进行了讲解。相信读者掌握了本章介绍的内容，就可以将 ECMAScript 脚本语言中运算符应用到具体开发实践中了。

第 7 章

流程控制语句

本章将介绍 ECMAScript 语法规范中关于流程控制语句的内容。ECMAScript 流程控制语句是脚本编程语言中的核心部分，主要包括条件语句、选择语句、循环迭代语句、循环中断语句及标签语句等。

7.1　if 条件语句

本节将介绍 ECMAScript 语法中最基本的 if 条件语句，它是使用频率非常高的流程控制语句。if 条件语句的语法形式有很多种，尤其与 else 关键字配合起来使用，表现形式更为多样。

7.1.1　if 语句

ECMAScript 语法规范中定义的 if 语句是最基本的条件选择语句，相当于“若……则……”的条件选择。

关于 if 语句的语法格式如下：

```
if(条件) {
    仅当条件为 true 时，执行此处代码
}
```

下面就是一个使用 if 条件选择语句的代码示例（详见源代码 ch07 目录中的 ch07-es-if.html 文件）。

【代码 7-1】

```
01  <script type="text/javascript">
02      if(true) {
03          console.log("if(true) {");
```

```
04            console.log("    to print true.");
05            console.log("}");
06        }
07        console.log("--------------------------------");
08        if(false)
09            console.log("if(false) {");
10            console.log("    to print false.");
11            console.log("}");
12    </script>
```

关于【代码 7-1】的分析如下：

第 02～06 行代码是第一个 if 语句块，主要是通过 if 语句判断 true 是否为真，如果为“真”则通过第 03～05 行代码控制输出调试信息。

第 08～11 行代码是意图模仿第 02～06 行代码的第二个 if 语句块，主要是通过 if 语句判断 false 是否为真，如果不为“真”则不执行第 09～11 行代码。但注意到第 08～11 行代码中语句块没有“{}”符号，那么这个模仿的第二个 if 语句块会不会和第一个 if 语句块功能完全一样呢？

运行页面，使用调试器查看控制台输出的调试信息，效果如图 7.1 所示。第 02～06 行代码的第一个 if 语句块输出的内容是符合预期的；第 08～11 行代码的第二个 if 语句块输出的内容则比较奇怪，原本由于 if 语句判断布尔值 fasle 后不会得到任何输出，但第 10～11 行代码的内容却输出了，而第 09 行代码的内容没有输出。这是由于第二个 if 语句块没有“{}”符号的缘故造成的，其默认只将第 09 行代码认作 if 语句块的语句体，而第 10～11 行代码不是其语句体，因此这两行代码的内容也就得到了输出。

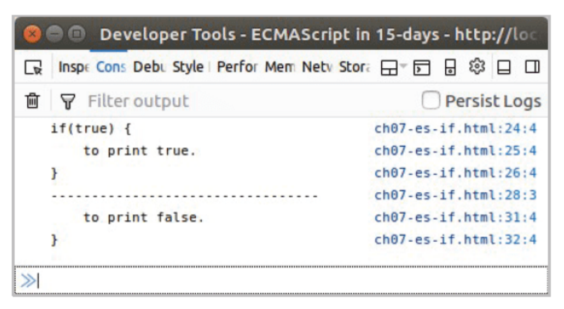


图 7.1　if 语句

7.1.2　if…else…语句

ECMAScript 语法规范中定义的“if…else…”语句是对 if 语句的增强，相当于“若…则…，否则…”条件选择。

关于 if…else…语句的语法格式如下：

```
if (条件) {
    仅当条件为 true 时，执行此处代码
} else {
    否则执行此处代码
}
```

下面是一段使用 if…else…语句的代码示例（详见源代码 ch07 目录中的 ch07-es-if-else.html 文件）。

【代码 7-2】

```
01   <script type="text/javascript">
02       if(true) {
03           console.log("if(true) {");
04           console.log("    true");
05           console.log("}");
06       } else {
07           console.log("else {");
08           console.log("    false");
09           console.log("}");
10       }
11      console.log("--------------------------------");
12       if(false) {
13           console.log("if(true) {");
14           console.log("    true");
15           console.log("}");
16       } else {
17           console.log("else {");
18           console.log("    false");
19           console.log("}");
20       }
21   </script>
```

关于【代码 7-2】的分析如下：

第 02～10 行代码是第一个 if…else…语句块，主要是通过 if 语句判断 true 是否为真，如果为“真”则执行 if 语句块中第 03～05 行的代码并在控制台输出调试信息，否则执行 else 语句块内的第 07～09 行的代码并在控制台输出调试信息。

第 12～20 行代码是第二个 if…else…语句块，主要是通过 if 语句判断 false 是否为真，然后执行 if 和 else 语句块内的代码。

运行页面，查看控制台输出的调试信息，如图 7.2 所示。对于 if…else…条件选择语句，运行中只能执行 if 语句块或 else 语句块中的内容，不能两者都执行，也不能都不执行，这也正是 if…else…条件选择语句的特点。

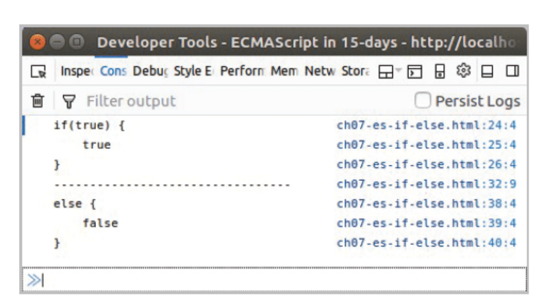


图 7.2　if…else…语句

7.1.3　if…else if…else…语句

ECMAScript 语法规范中定义的 if…else if…else…语句是条件选择语句的最完整的版本了，相当于“如果…则…，如果…则…，否则…”条件选择，基本可以适用于编程中遇到的任何场景。

关于 if…else if…else…语句的语法格式如下：

```
if (条件 1) {
    仅当条件 1 为 true 时，执行此处代码
} else if (条件 2) {
    仅当条件 2 为 true 时，执行此处代码
} … {
 …
} else if (条件 n) {
    仅当条件 n 为 true 时，执行此处代码
} else {
    否则执行此处代码
}
```

下面是一段使用 if…else if…else…语句的代码示例（详见源代码 ch07 目录中的 ch07-es-if-else-if.html 文件）。

【代码 7-3】

```
01  <script type="text/javascript">
02      if(true) {
03          console.log("1- if");
04      } else if (false) {
05          console.log("1- else if");
06      } else {
07          console.log("1- else");
08      }
09      if(false) {
10          console.log("2 - if");
11      } else if (true) {
12          console.log("2 - else if");
13      } else {
14          console.log("2 - else");
15      }
16      if(false) {
17          console.log("3 - if");
18      } else if (false) {
19          console.log("3 - else if");
20      } else {
21          console.log("3 - else");
22      }
23  </script>
```

关于【代码 7-3】的分析如下：

这段代码主要是使用了 3 段 if…else if…else…语句块，分别用于演示 if 语句块、else if 语句块和 else 语句块 3 种不同的条件输出。

第 02～08 行代码是第一个 if…else if…else…语句块，主要是通过 if 语句判断 true 是否为真，如果为“真”则执行 if 语句块中第 03 行的代码并在控制台输出调试信息，否则执行后面的语句。

第 09～15 行代码是第二个 if…else if…else…语句块，主要是通过 if 语句判断 false 是否为真，如果不为“真”则继续通过 else if 语句判断 true 是否为真，如果为“真”则执行 else if 语句块中第 12 行的代码并在控制台输出调试信息，否则执行后面的语句。

第 16～22 行代码是第三个 if…else if…else…语句块，主要是通过 if 语句判断 false 是否为真。

如果不为“真”，则继续通过 else if 语句判断 false 是否为真；如果为“真”，则执行 else 语句块中第 21 行的代码并在控制台输出调试信息。

页面效果如图 7.3 所示。对于 if…else if…else…条件选择语句，运行中只能执行 if 语句块、else if 语句块或 else 语句块中的内容，不能全部都执行，也不能都不执行，至少执行一个语句块，这也同样是 if…else if…else…条件选择语句的特点。

另外，else if 语句块可以扩展为多项并列的形式，这样就可以适用于绝大多数的编程场景了。当然，如果并列项太多，就可以使用下面我们将要介绍的 switch 条件选择语句。

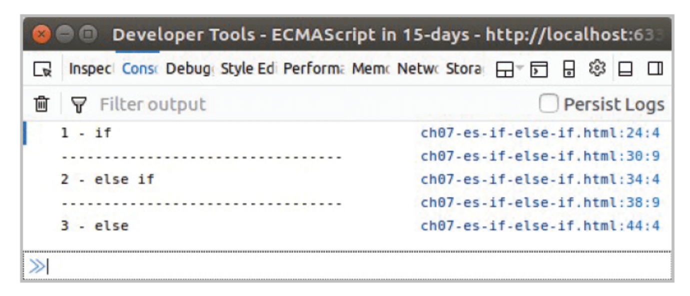


图 7.3　if…else if…else…语句

7.2　switch 条件语句

本节将介绍 ECMAScript 语法规范中另一个条件选择语句——switch 语句。在大多数情况下，比如选择项很多的编程场景，switch 条件选择语句就会比 if 条件选择语句更为适用。

ECMAScript 语法规范中定义的 switch 语句同样是一种条件选择语句，主要用于基于不同的条件执行不同的操作的场景。

关于 switch 语句的语法格式如下：

```
switch(n) {
  case 1:
    执行代码块 1;
    break;
  case 2:
    执行代码块 2;
    break;
……
  case n:
    执行代码块 n;
    break;
  default:
     与 case 1、 case 2、...、 case n 不同时要执行的代码;
}
```

其中，n 是用于选择的表达式（通常是一个变量）。表达式的值依次与结构体中的每一个 case 值进行比较，如果存在匹配的 case 项，则执行该 case 项的代码块；如果不存在任何一个匹配的 case 项，则执行 default 项的代码块。另外，case 项与 case 项之间通过 break 来间隔，default 项通常写

在全部 case 项之后。

下面看一段使用 switch 语句的代码示例(详见源代码 ch07 目录中的 ch07-es-switch.html 文件)。

【代码 7-4】

```
01  <!-- 添加文档主体内容 -->
02  前端编程语言菜单:  
03  <select id="id-select-switch" onchange="on_select_change(this.value);">
04      <option value="ECMAScript">ECMAScript</option>
05      <option value="JavaScript">JavaScript</option>
06      <option value="TypeScript">TypeScript</option>
07  </select>
08  <div id="id-div-switch">
09      请选择您喜欢的前端编程语言:
10  </div>
11  <!-- 添加文档主体内容 -->
12  <script type="text/javascript">
13      var v_div = document.getElementById("id-div-switch");
14      function on_select_change(value) {
15          switch(value) {
16              case "ECMAScript":
17                  v_div.innerHTML = "您选择前端编程语言: " + value;
18                  break;
19              case "JavaScript":
20                  v_div.innerHTML = "您选择前端编程语言: " + value;
21                  break;
22              case "TypeScript":
23                  v_div.innerHTML = "您选择前端编程语言: " + value;
24                  break;
25              default:
26                  v_div.innerHTML = "您选择前端编程语言: ";
27                  break;
28          }
29      }
30  </script>
```

关于【代码 7-4】的分析如下:

第 03～07 行代码通过<select>标签定义了一个下拉选择框控件，并添加了 3 个<option>选择项。其中，第 03 行代码中为该标签定义了“id”属性，并定义了“onchange”事件处理函数方法（on_select_change(this.value)），参数“this.value”代表<select>标签的选中值。有关于 JavaScript 事件处理的内容，我们会在后续章节中详细介绍，此处只需要知道“onchange”事件是用户操作<select>标签后被触发的就可以了。

第 09～11 行代码通过<div>标签定义了一个层，用于显示用户操作<select>标签结果的返回值。

第 12～30 行代码通过<script>标签定义了一段嵌入式 JavaScript 脚本。

第 13 行代码通过 document.getElementById()方法获取了第 08～10 行代码定义的<div>标签的“id”值。

第 14～29 行代码定义了第 03 行代码中的“onchange”事件处理函数(on_select_change(value))，参数“value”为传递过来的<select>标签的选中值。

第 15～28 行代码通过 switch 语句对参数“value”进行了选择判断，其中每个 case 语句定义了根据不同选择所执行的代码，主要是通过“innerHTML”属性将用户的操作结果显示到第 09～

11 行代码定义的<div>标签中。

页面初始效果如图 7.4 所示。单击下拉菜单后任意选择一项，操作后的页面效果如图 7.5 和 7.6 所示。



图 7.4　switch 语句（1）

图 7.5　switch 语句（2）



图 7.6　switch 语句（3）

7.3　循环迭代语句

本节将介绍 ECMAScript 语法规范中的循环迭代语句。循环语句和迭代语句其实是一个意思，就是声明一组要反复执行的命令，直到满足某些条件为止。由于循环条件通常为用于迭代的值或是执行重复的算术任务，因此命名为循环迭代语句。

ECMAScript 语法规范中定义的循环迭代语句与其他高级编程语言（如 C 语言、Java 语言等）类似，具体包括 for 语句、while 语句、do…while…语句、for…in…语句及中断语句等，下面我们将逐一介绍。

7.3.1　for 语句

ECMAScript 语法规范中定义的 for 语句是循环语句，主要用于一次次的循环重复执行相同的代码，且每次执行代码时的自变量参数会按照规律递增或递减。

关于 for 语句的语法格式如下：

```
for (语句 1; 语句 2; 语句 3) {
  被执行的代码块
}
```

其中，语句 1 是在循环（代码块）开始前执行，一般用于定义自变量参数初始条件；语句 2 定义运行循环（代码块）的条件，一般用于定义自变量参数结束条件；语句 3 在循环（代码块）已被执行之后执行，一般用于定义自变量变化规律。

下面是一段使用 for 语句的代码示例（详见源代码 ch07 目录中的 ch07-es-for.html 文件）。

【代码 7-5】

```
01  <script type="text/javascript">
02   for(var i=1; i<=3; i++) {
03       console.log("i=", i);
04   }
05  </script>
```

关于【代码 7-5】的分析如下：

第 02～04 行代码通过 for 语句定义了一个循环体。其中，第 02 行代码定义了 for 语句的循环初始条件（var i=1;）、循环结束条件（i<=3;）及自变量的变化规律（i++）。该 for 循环相当于循环执行了 3 次第 03 行代码定义的控制台调试信息输出功能。

页面效果如图 7.7 所示。

如果将 for 语句嵌套起来使用，就可以实现很多既复杂又有趣的功能，如九九乘法表的打印。下面就介绍一下如何使用 for 语句实现打印九九乘法表的代码示例（详见源代码 ch07 目录中的 ch07-es-for-9x9.html 文件）。

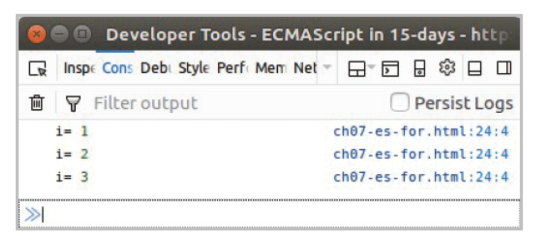


图 7.7　for 语句

【代码 7-6】

```
01  <script type="text/javascript">
02      for(var i=1; i<=9; i++) {
03          var v_line = "";
04          for(var j=1; j<=i; j++) {
05              v_line += j + "x" + i + "=" + j*i + " ";
06          }
07          console.log(v_line);
08      }
09  </script>
```

关于【代码 7-6】的分析如下：

这段代码主要就是通过嵌套使用 for 循环语句（双层 for 循环）来实现九九乘法表的打印。

第 02～08 行代码通过 for 语句定义了第一层循环体（或称外层循环体）。其中，自变量定义为“i”，循环次数是 9。

第 04～06 行代码通过 for 语句定义了第二层循环体（或称内层循环体）。其中，自变量定义为“j”，循环次数是依据变量 i 的取值定义的；第 05 行代码用于保存九九乘法表的每一行。

第 07 行代码通过调试信息在控制台输出九九乘法表。

页面效果如图 7.8 所示。

图 7.8　for 语句实现九九乘法表输出

7.3.2　for…in…语句

在 ECMAScript 语法规范中，在 for 语句的基础上又定义了一个 for…in…迭代语句，for…in…语句是更严格的迭代语句，主要用于枚举对象集合中的属性。

关于 for…in…迭代语句的语法格式如下：

```
for (prop in collection) {
  被执行的代码块
}
```

其中，prop 用于表示属性的变量，collection 用于表示属性的集合。

下面是一段使用 for…in…语句的代码示例（详见源代码 ch07 目录中的 ch07-es-for-in.html 文件）。

【代码 7-7】

```
01  <script type="text/javascript">
02      var arr = new Array();
03      for(var i=1; i<=3; i++) {
04          arr[i] = i;
05      }
06      var j;
07      for(j in arr) {
08          console.log("j = " + j);
09      }
10  </script>
```

关于【代码 7-7】的分析如下：

第 02 行代码定义了一个数组变量 arr，并通过第 03～05 行代码中 for 语句进行了初始化赋值（数字 1～3）。

第 06 行代码定义了一个变量 j，用于 for…in…语句中的变量。

第 07～09 行代码中通过 for…in…语句迭代了数组变量 arr 中每一个属性，注意 for…in…语句的书写方法 for(j in arr)。

页面效果如图 7.9 所示。

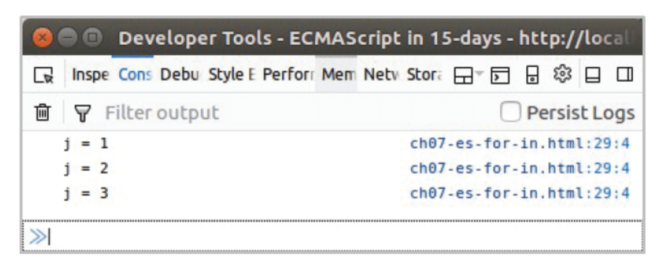


图 7.9　for…in…语句

如前文所述，for…in…语句主要用来迭代对象中的属性，下面是一段使用 for…in…语句迭代 window 对象中属性的代码示例（详见源代码 ch07 目录中的 ch07-es-for-in-window.html 文件）。

【代码 7-8】

```
01  <script type="text/javascript">
02      var wProp;
03      for(wProp in window) {
04          console.log("obj in window = " + wProp);
05      }
06  </script>
```

关于【代码 7-8】的分析如下：

第 02 行代码定义了一个变量 wProp，用于 for…in…语句中的变量。

第 03～05 行代码中通过 for…in…语句迭代了 window 对象中的属性（for(wProp in window)），并在控制台中进行了输出。

页面效果如图 7.10 所示。在浏览器控制台中输出了 window 对象中的一部分属性（window 对象中有很多属性），读者看看这一小部分属性名是不是觉得很熟悉的呢？

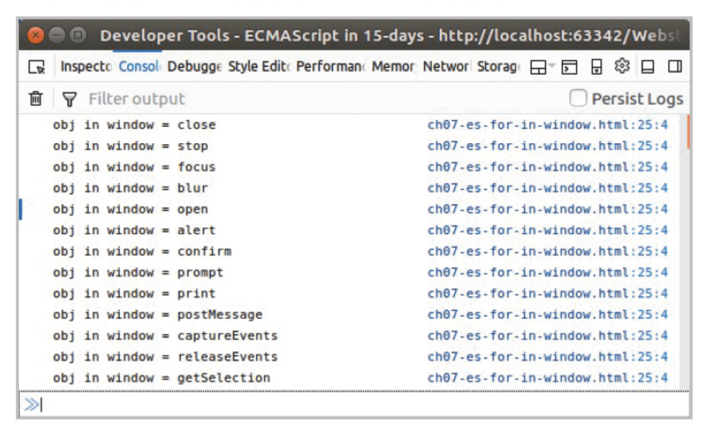


图 7.10　for…in…语句迭代 window 对象属性

7.3.3　while 语句

在 ECMAScript 语法规范中还定义了一个 while 循环迭代语句。在前文中介绍的 for 语句用于定义有穷循环语句，而 while 语句则用来定义无穷循环语句（需要有循环结束条件的）。

关于 while 语句的语法格式如下：

```
while (条件) {
  被执行的代码块
}
```

其中，当判断“条件”为真时则无限次执行循环体内的代码，只有当判断“条件”为假时才停止循环，因此 while 是一种前测试循环语句。

下面是一段使用 while 语句来实现的代码示例（详见源代码 ch07 目录中的 ch07-es-while.html 文件），这段代码是将【代码 7-5】中的 for 语句通过 while 语句改写而成。

【代码 7-9】

```
01  <script type="text/javascript">
02   var i = 1;
03   while(i <= 3) {
04        console.log("i=", i);
05        i++;
06   }
07  </script>
```

关于【代码 7-9】的分析如下：

第 02 行代码定义了一个变量 i，并进行了初始化操作 i=1。

第 03～06 行代码通过 while 语句定义了一个循环体。其中，第 03 行代码通过 while 语句判断变量 i 的值是否小于等于数值 3，若条件为“真”则执行第 04 和第 05 行代码定义的循环体，且第 05 行代码执行变量 i 的自动累加。

当第 05 行代码被执行后，变量 i 的累加值大于数值 3 时，第 03 行代码的判断条件为“假”，从而结束第 03～06 行代码定义的 while 循环体。

页面效果如图 7.11 所示。

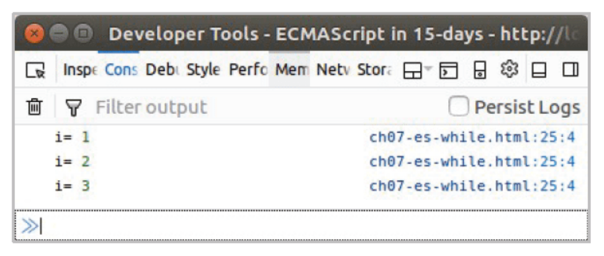


图 7.11　while 语句

7.3.4　do…while 语句

前文中介绍的 while 语句是前测试循环语句，与之对应的则是 do…while 后测试循环语句，即退出条件在执行循环内部的代码之后计算，也就是表示在判断测试条件之前，至少会执行循环主体一次。

关于 do...while 语句的语法格式如下：

```
do {
  被执行的代码块
} while (条件);
```

其中，当判断“条件”为真时则无限次执行循环体内的代码，只有当判断“条件”为假时才停止循环。

下面是一段使用 do...while 语句来实现的代码示例（详见源代码 ch07 目录中的 ch07-es-do-while.html 文件）。

【代码 7-10】

```
01  <script type="text/javascript">
02   var i = 1;
03   do {
04        console.log("i=", i);
05        i++;
06   } while(i <= 3);
07  </script>
```

关于【代码 7-10】的分析如下：

第 02 行代码定义了一个变量 i，并进行了初始化操作 i=1。

第 03～06 行代码通过 do...while 语句定义了一个循环体。其中，第 03 行代码中使用 do 语句先执行循环体；第 05 行代码执行变量 i 的自动累加；第 06 行代码通过 while 语句判断变量 i 的值是否小等于数值 3，如果条件为“真”则执行第 04～05 行代码定义的循环体。

当第 05 行代码被执行后，变量 i 的累加值大于数值 3 时，第 03 行代码的判断条件为“假”，从而跳出第 03～06 行代码定义的 while 循环体。

页面效果如图 7.12 所示。

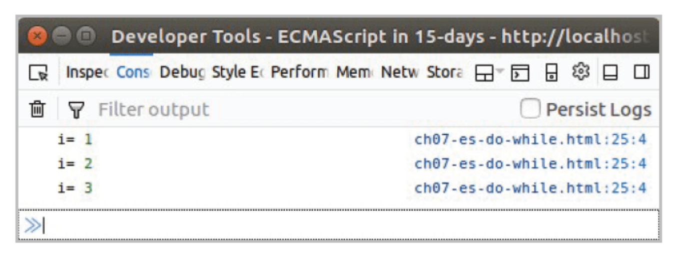


图 7.12　do...while 语句

7.4　循环中断语句

本节将介绍 ECMAScript 语法规范中的循环中断语句。所谓循环中断语句，就是终止循环语句继续执行的语句。在 ECMAScript 语法规范中定义的循环中断语句主要是指 break 语句和 continue 语句，使用循环中断语句可以对循环中的代码执行提供更严格的控制。

7.4.1 break 语句

ECMAScript 语法规范中提供了一个 break 语句，用来实现根据指定条件结束循环体的功能。具体来说就是 break 语句可以实现跳出循环体的功能，且跳出循环体后不影响后续代码的执行（如果有后续代码的话）。

关于 break 语句的语法格式如下：

```
循环体 {
  break;
}
```

下面是一段使用 break 语句终止循环体的代码示例（详见源代码 ch07 目录中的 ch07-es-break.html 文件）。

【代码 7-11】

```
01  <script type="text/javascript">
02   for(var i=1; i<=6; i++) {
03        if(i > 5)
04             break;
05        console.log("i=", i);
06   }
07  </script>
```

关于【代码 7-11】的分析如下：

这段代码主要就是对 for 语句循环体设定了提前结束循环的条件，具体是通过第 03 和第 04 行代码定义的 if 语句，判断自变量 i 是否大于数值 5，若条件为真，则执行第 04 行代码定义的 break 语句终止循环。

页面效果如图 7.13 所示。

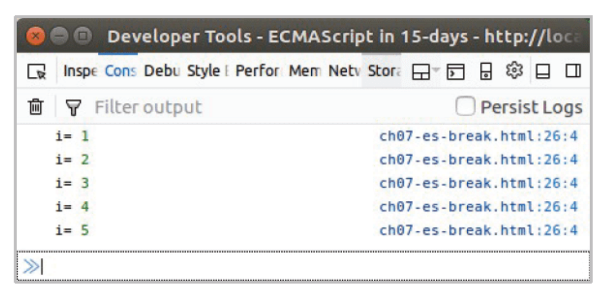


图 7.13 break 语句

7.4.2 continue 语句

虽然 break 语句功能强大，但也有其局限性，因为中断循环后后续的循环也会随之被终止。如果想仅仅中断一次循环该如何操作呢？

在 ECMAScript 语法规范中提供了一个 continue 语句，用于中断循环体中的一次迭代。具体来讲，就是在一次循环中即使满足了中断的条件，也仅仅是结束本次循环，不影响循环体继续后续的迭代。

关于 continue 语句的语法格式如下：

```
循环体 {
  continue;
}
```

下面是一段使用 continue 语句终止循环体的代码示例（详见源代码 ch07 目录中的 ch07-es-continue.html 文件）。

【代码 7-12】

```
01  <script type="text/javascript">
02      for(var i=1; i<=9; i++) {
03          if((i == 1) || (i == 4) || (i == 7))
04              continue;
05          console.log("i=", i);
06      }
07  </script>
```

关于【代码 7-12】的分析如下：

这段代码主要就是对 for 语句循环体设定了终止本次循环的条件。第 03 和第 04 行代码定义了一个 if 语句，判断自变量 i 是否等于数值 1、4 和 7，若条件为真，则执行第 04 行代码定义的 continue 语句中断本次循环。

页面效果如图 7.14 所示。当循环执行到自变量为 1、4 和 7 时被中断了，第 05 行代码定义的调试信息输出没有被执行，而从显示的结果来看，其他数值均在控制台得到了有效的打印输出。

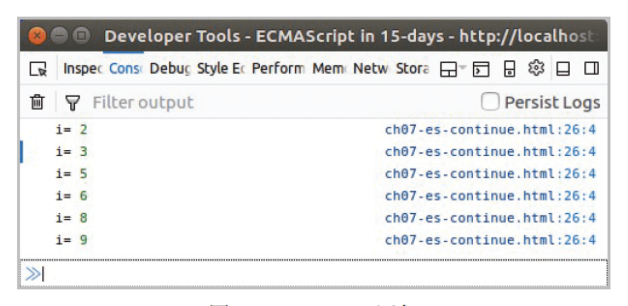


图 7.14　continue 语句

7.4.3　break 语句与标签语句配合使用

在 ECMAScript 语法规范中还提供了一个标签语句，可以与中断语句配合使用，用以返回到代码中的特定位置。

下面是一段 break 语句与标签语句配合使用的代码示例（详见源代码 ch07 目录中的 ch07-es-break-label.html 文件）。

【代码 7-13】

```
01  <script type="text/javascript">
02      label:
03      for(var i=1; i<=9; i++) {
04          var v_line = "";
05          for(var j=1; j<=i; j++) {
```

```
06                if((i>6) && (j>6))
07                    break label;
08                v_line += j + "x" + i + "=" + j*i + " ";
09            }
10            console.log(v_line);
11        }
12    </script>
```

关于【代码 7-13】的分析如下：

这段代码其实是对【代码 7-6】中实现的 9×9 乘法表进行了改写，目的就是想提前结束循环，即当乘法表打印到第 6 行时强行中断该循环体。

因此，第 02 行代码定义了一个标签 label；第 06 和第 07 行代码定义了一个 if 语句，判断自变量 i 和 j 是否大于数值 6，若条件为真，则执行 break 语句（break label）终止循环并跳转到标签 label 处。

页面效果如图 7.15 所示。如图 7.15 中箭头所指，当循环体执行到第 6 行时，循环体被终止执行，说明第 07 行代码定义的 break 语句（break label）起到了作用。

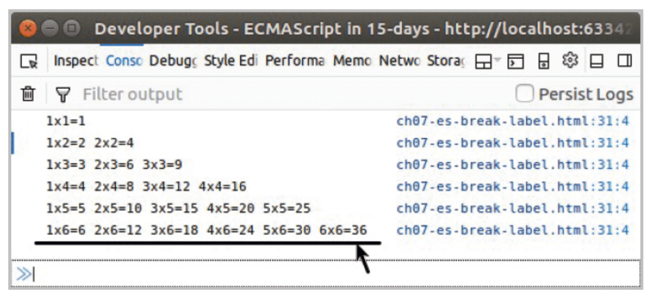


图 7.15　break 语句与标签语句（1）

如果仔细研究【代码 7-13】会发现，即使不使用标签语句（仅使用 break 语句），效果与使用标签 label 也是一样的。

下面是一段真正发挥标签语句作用的代码示例（详见源代码 ch07 目录中的 ch07-es-break-label-spec.html 文件）。

【代码 7-14】

```
01    <script type="text/javascript">
02        for(var i=1; i<=9; i++) {
03            var v_line = "";
04            label:
05            for(var j=1; j<=i; j++) {
06                if((i>6) && (j>6))
07                    break label;
08                v_line += j + "x" + i + "=" + j*i + " ";
09            }
10            console.log(v_line);
11        }
12    </script>
```

关于【代码 7-14】的分析如下：

这段代码是对【代码 7-13】进行了改写，目的就是想测试一下标签 label 放到不同位置的效果。

因此，将【代码 7-13】中第 02 行代码定义的标签 label 重新放到了【代码 7-14】中第 04 行的位置，其他代码没有做任何改动。

页面效果如图 7.16 所示。9×9 乘法表的第 7～9 列打印时被截断了，没有全部打印出来。但与图 7.15 效果不同的是，乘法表打印到第 6 行后没有被终止，仍是继续打印出了第 7～9 行的乘法表。原因就是我们将标签 label 的位置改到了代码的第 04 行，即放在了外层循环体内部、内层循环体的外部。这样当 break 语句（break label）被执行后跳转到第 04 行的标签 label 位置时，仅仅是终止了内层循环体，而外层循环体会继续执行下去。

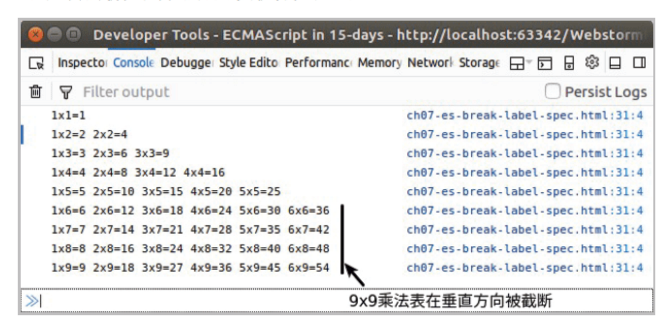


图 7.16　break 语句与标签语句（2）

7.4.4　continue 语句与标签语句配合使用

7.4.3 小节介绍了 break 语句和标签语句配合使用的效果，本小节再介绍一下 continue 语句和标签语句配合使用。

下面是一段 continue 语句与标签语句配合使用的代码示例（详见源代码 ch07 目录中的 ch07-es-continue-label.html 文件）。

【代码 7-15】

```
01  <script type="text/javascript">
02      label:
03      for(var i=1; i<=9; i++) {
04          var v_line = "";
05          for(var j=1; j<=i; j++) {
06              if((i>6) && (j>6))
07                  continue label;
08              v_line += j + "x" + i + "=" + j*i + " ";
09          }
10          console.log(v_line);
11      }
12  </script>
```

关于【代码 7-15】的分析如下：

这段代码是对【代码 7-13】中实现的 9×9 乘法表进行了改写，将 break 语句换成了 continue 语句。

页面效果如图 7.17 所示。【代码 7-15】中使用 continue 的效果与【代码 7-13】中使用 break 的效果是一致的。但仔细研究一下会发现，若【代码 7-15】中不使用标签语句，则 continue 语句

会使得外层循环体继续执行下去，这也正是标签语句 label 起到了终止循环体的作用。

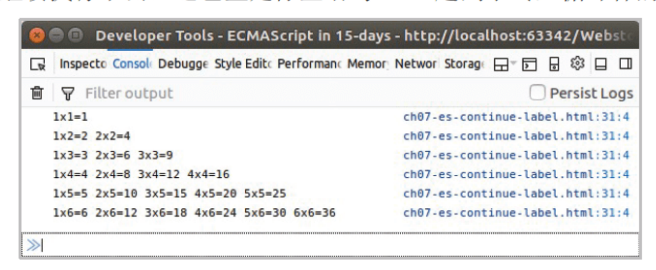


图 7.17　continue 语句与标签语句（1）

如果继续将【代码 7-14】中的 break 语句改写成 continue 语句呢？

下面是一段 continue 语句与标签语句配合的代码示例（详见源代码 ch07 目录中的 ch07-es-continue-label-spec.html 文件）。

【代码 7-16】

```
01  <script type="text/javascript">
02      for(var i=1; i<=9; i++) {
03          var v_line = "";
04          label:
05          for(var j=1; j<=i; j++) {
06              if((i>6) && (j>6))
07                  continue label;
08              v_line += j + "x" + i + "=" + j*i + " ";
09          }
10          console.log(v_line);
11      }
12  </script>
```

关于【代码 7-16】的分析如下：

这段代码是对【代码 7-15】进行了改写，目的就是想测试一下标签 label 放到不同位置的效果。

因此，将【代码 7-15】中第 02 行代码定义的标签 label 重新放到了【代码 7-16】中第 04 行的位置，其他代码没有做任何改动。

页面效果如图 7.18 所示。9×9 乘法表的第 7～9 列在打印时同样被截断了，没有全部打印出来，这与图 7.16 的效果是一致的。

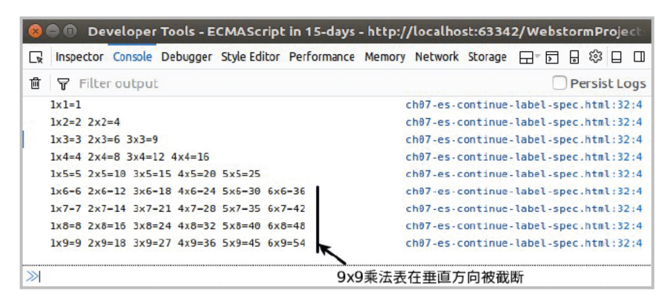


图 7.18　continue 语句与标签语句（2）

7.5　ECMAScript 6 新特新——for of 迭代循环

ECMAScript 6 语法规范中新定义了一个 for of 循环迭代语句，专门用来针对可迭代的 JavaScript 对象执行迭代操作。下面通过几个简单的代码实例，介绍一下 for of 循环迭代语句的使用方法。

7.5.1　迭代数组

for of 循环迭代语句主要用于迭代数组。下面是一段使用 for of 循环迭代语句迭代数组的代码示例（详见源代码 ch07 目录中的 ch07-es-for-of-arr.html 文件）。

【代码 7-17】

```
01  <script type="text/javascript">
02      var arr = new Array();
03      for (let i = 0; i <= 3; i++) {
04          arr[i] = i;
05      }
06      for (let j of arr) {
07          console.log("j = " + j);
08      }
09  </script>
```

关于【代码 7-17】的分析如下：

第 02 行代码定义了一个数组变量 arr，并通过第 03～05 行代码中 for 语句进行了初始化赋值（数字 0～3）。

第 06～08 行代码中通过 for…of…循环迭代语句对数组变量 arr 进行了迭代操作，并将结果输出到浏览器控制台中。

页面效果如图 7.19 所示。通过 for…of…循环迭代语句成功对数组变量 arr 进行了迭代操作。

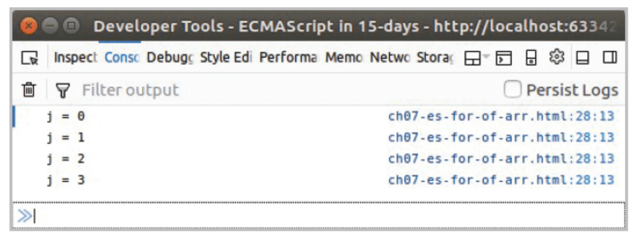


图 7.19　for…of…循环迭代语句操作数组

7.5.2　迭代字符串

使用 for of 循环迭代语句另一个常见的应用场景就是迭代字符串了。下面是一段使用 for of 循环迭代语句操作字符串的代码示例（详见源代码 ch07 目录中的 ch07-es-for-of-string.html 文件）。

【代码 7-18】

```
01  <script type="text/javascript">
02      var str = new String("abcdef");
03      console.log("--- for of 'abcde' ---");
04      for (let c of str) {
05          console.log(c);
06      }
07  </script>
```

关于【代码 7-18】的分析如下：

第 02 行代码定义了一个字符串变量 str，并初始化为 "abcdef"。

第 04～06 行代码中通过 for…of…循环迭代语句对字符串变量 str 进行了迭代操作，并将结果输出到浏览器控制台中。

页面效果如图 7.20 所示。通过 for…of…循环迭代语句成功对字符串变量进行了迭代操作，返回结果为单个字符的序列。

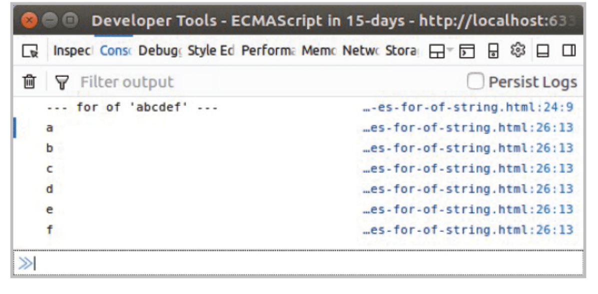


图 7.20　for…of…循环迭代语句操作字符串

7.5.3　for of 循环迭代原理

这里有必要再简单介绍一下 for of 循环迭代的原理，其实 ECMAScript 6 语法规范中新定义的 for of 循环迭代语句是通过迭代器 Symbol.iterator 方法来实现的。迭代器 Symbol.iterator 中定义了一个 iterator.next()方法，用于依次遍历迭代对象中的每一个值，直到遍历完成才会退出，而 for of 循环迭代语句就是通过该机制实现了循环迭代操作。感兴趣的读者可以阅读一下官方的 ECMAScript 6 语法规范文档。

既然 for of 循环迭代语句是通过迭代器 Symbol.iterator 方法来实现的，那么被迭代的对象就是有要求的，并不是所有 ECMAScript 定义的对象都可以被 for of 语句操作。可被 for of 循环迭代语句操作的对象主要有数组、字符串、Set 对象和 Map 对象。若尝试操作其他类型的对象，则很可能会抛出异常。

7.6　本章小结

本章主要介绍了 ECMAScript 语法中流程控制语句的内容，具体包括 if 和 switch 条件选择语句、for 和 while 循环迭代语句、break 和 continue 循环中断语句及其与标签语句配合使用等，并通过一些具体实例进行了讲解。相信读者掌握了本章介绍的内容，就可以使用 ECMAScript 脚本语言进行功能更为复杂的开发实践了。

第 8 章

函　数

本章将介绍 ECMAScript 语法规范中关于函数的内容，函数在 ECMAScript 语法规范中属于有一定难度的知识点。本章内容主要包括函数定义、函数调用、函数对象等。

8.1　ECMAScript 函数基础

首先介绍一下计算机程序设计语言中关于“函数”的概念。所谓“函数”其实就是从英文单词“function”翻译过来的。因此，绝大多数程序设计语言中的函数也是通过“function”关键字来定义的。

函数的功能是什么呢？简单来说，“函数”就是用来完成特定功能的程序语句集合。传统高级程序设计语言（如 C、C#、Java 等）中的函数，一般都是通过关键字声明和定义，然后通过调用来使用的。

而对于将函数作为一个参数传给另一个函数，或者是赋值给一个本地变量，又或者是作为返回值进行返回这样的高级用法，都需要通过函数指针（function pointer）或代理（delegate）的方式来实现。

在 ECMAScript 语法规范中，函数不仅可以像传统函数的使用方式（声明、定义和调用）一样，还可以像简单值一样进行赋值、传递参数及返回值的操作。因此，ECMAScript（JavaScript）函数也被称为“第一类函数（First-class Function）”。进一步来说，ECMAScript（JavaScript）函数既实现了像类（Class）的构造函数一样的作用，又是一个 Function 类的实例（instance）。

8.2　ECMAScript 函数声明、定义与调用

本节将介绍 ECMAScript 函数的基本使用方法，具体就是 ECMAScript 函数声明、定义与调用等方面的内容。在 ECMAScript 语法规范中，函数声明与定义的方式非常灵活，ECMAScript 语法规范为设计人员实现了多种函数声明与定义的方式。

8.2.1　传统方式定义 ECMAScript 函数

这里先介绍传统方式下如何声明与定义 ECMAScript 函数。传统方式对于绝大多数的高级程序设计语言都是通用的，具体的语法格式如下：

```
function 函数名(参数1, 参数2, ...) {
    // 函数体内定义的语句
}
```

下面是一个使用这种声明定义方式（无函数参数）的代码示例（详见源代码 ch08 目录中的 ch08-es-func-define-basic.html 文件）。

【代码 8-1】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——传统方式
04       */
05      function MessageBox() {
06          alert("传统的 ECMAScript 函数声明与定义方式（无函数参数）");
07      }
08      MessageBox(); // 调用 ECMAScript 函数
09  </script>
```

关于【代码 8-1】的分析如下：

第 05～07 行代码通过 function 关键字声明定义了函数，函数名为 MessageBox()，这就是传统的高级程序语言声明定义方式。其中，函数体通过大括号“{}”来定义，第 06 行代码为函数体内定义的语句。

第 08 行代码直接通过函数名 MessageBox() 的方式调用函数。

页面效果如图 8.1 所示。【代码 8-1】中第 05～07 行代码定义的函数 MessageBox()被成功调用了，弹出一个信息警告框。

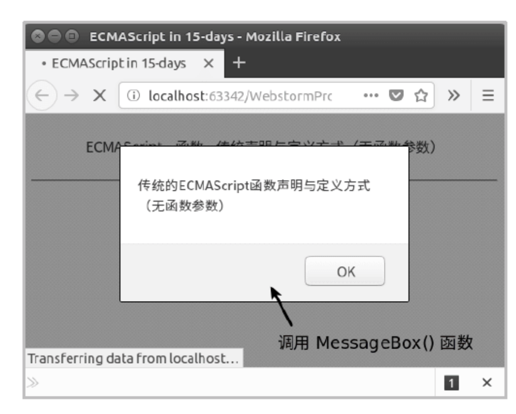


图 8.1　传统 ECMAScript 函数的声明与定义方式（1）

下面继续看一个使用这种声明定义方式（带参数）的代码示例（详见源代码 ch08 目录

中的 ch08-es-func-define-basic-param.html 文件）。

【代码 8-2】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——传统方式
04       */
05       function MessageBox(param1, param2) {
06           alert(param1 + " " + param2 + "!");
07       }
08       MessageBox("Hello", "ECMAScript");   // 调用 ECMAScript 函数
09  </script>
```

关于【代码 8-2】的分析如下：

第 05～07 行代码通过 function 关键字声明定义了函数，函数名为 MessageBox(param1, param2)，该函数定义了两个参数 param1 和 param2。

第 08 行代码直接通过函数名 MessageBox("Hello", "ECMAScript")调用函数，注意传递的两个参数值"Hello"和"ECMAScript"。

页面效果如图 8.2 所示。第 09 行代码调用的函数 MessageBox()中所定义的两个参数"Hello"和"ECMAScript"被成功传递了。

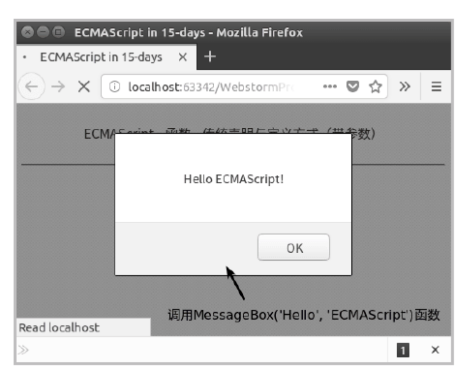


图 8.2　传统 ECMAScript 函数的声明与定义方式（2）

8.2.2　ECMAScript 函数表达式方式

表达式是高级语言程序设计中一个基本的概念，可能没有人会想到将函数也写成表达式的方式。不过，在 ECMAScript 语法规范中却支持将函数声明定义为表达式的方式，这也正是 ECMAScript 脚本语言与众不同之处。ECMAScript 脚本语言之所以有函数表达式的概念，是源自于 ECMAScript 语法规范中“一切均是对象”的设计理念。

关于函数表达式的基本语法格式如下：

```
var 函数名 = function(参数 1, 参数 2, ...){
    // 函数体内定义的语句
};
```

其中，函数名是函数声明语句必需的部分，其用途就如同变量一样，后面定义的函数对象会

赋值给这个变量。另外，function 关键字后面的函数名是可选的，即使加上该函数名也不是前面传统声明定义方式中的函数名了，二者功能完全不一样。

下面是一个使用 ECMAScript 函数表达式方式的代码示例（详见源代码 ch08 目录中的 ch08-es-func-define-exp.html 文件）。

【代码 8-3】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——函数表达式
04       */
05      var vSum = function(n1, n2) {
06          return n1 + n2;         // 返回值
07      };
08      console.log("vSum(3, 6) = " + vSum(3, 6));
09  </script>
```

关于【代码 8-3】的分析如下：

第 05～07 行代码通过 var 关键字定义了一个变量 vSum，同时后面通过 function 关键字声明定义了函数（注意没有定义函数名），这就是使用函数表达式定义函数的方式。在 function 关键字内定义了两个参数 n1 和 n2，用作两个运算数。函数体同样通过大括号“{}”来定义。第 06 行代码为函数体内定义的语句，通过 return 关键字返回两个参数 n1 和 n2 的算术和，这与传统声明定义函数的方法基本一致。

第 08 行代码直接通过变量名 vSum(3, 6)调用函数。

运行【代码 8-3】所定义的 HTML 页面，并使用调试器查看控制台输出的调试信息，页面效果如图 8.3 所示。调用 ECMAScript 函数表达式定义的函数与调用传统 ECMAScript 声明方式定义的函数在用法上略有不同（见【代码 8-3】中第 08 行代码）。

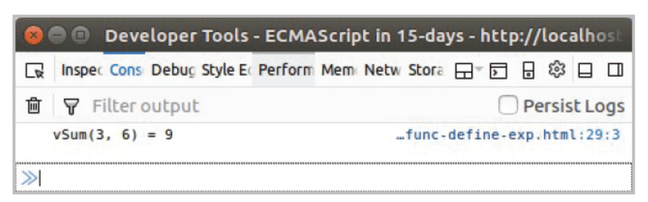


图 8.3　ECMAScript 函数表达式方式（1）

那么读者可能会有疑问了，ECMAScript 函数表达式方式的函数名呢？为了解答这个疑问，下面继续看一个使用函数表达式方式声明函数的代码示例（详见源代码 ch08 目录中的 ch08-es-func-define-exp-fname.html 文件），该函数带有函数名。

【代码 8-4】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——函数表达式
04       */
05      var vSum = function Sum(n1, n2) {
06          return n1 + n2;         // 返回值
07      };
08      console.log("Sum(3, 6) = " + Sum(3, 6));
```

```
09   </script>
```

关于【代码 8-4】的分析如下：

第 05～07 行代码通过 var 关键字定义了一个变量 vSum，同时后面通过 function 关键字声明定义了函数，注意这里添加了函数名 Sum。

第 08 行代码直接通过函数名 Sum(3, 6)调用函数。

页面效果如图 8.4 所示。页面并没有出现我们想要的效果，只是提示了函数名 Sum 未定义，也就是说【代码 8-4】中第 05 行代码定义的 Sum 不是一个有效的函数名。根据 ECMAScript 语法规范的定义，函数表达式只能通过前面定义的变量来调用。



图 8.4　ECMAScript 函数表达式方式（2）

那么，【代码 8-4】中定义的函数名 Sum 到底是什么呢？

为了解答这个疑问，下面改写一下【代码 8-4】（详见源代码 ch08 目录中的 ch08-es-func-define-exp-fname-is.html 文件）。

【代码 8-5】

```
01   <script type="text/javascript">
02       /*
03        * 定义 ECMAScript 函数——函数表达式
04        */
05       var vSum = function Sum(n1, n2) {
06           console.log("call Sum in func : " + Sum);
07           return n1 + n2;          // 返回值
08       };
09       console.log("vSum(3, 6) = " + vSum(3, 6));
10       console.log("call Sum out of func : " + Sum);
11   </script>
```

关于【代码 8-5】的分析如下：

第 06 行代码使用 console.log()方法在控制台输出 Sum 名称，注意这里是在函数体的内部。

第 09 行代码直接通过函数名 vSum(3, 6)调用函数，目的是为了让函数表达式 vSum 生效，并执行到第 06 行代码。

第 10 行代码再次使用 console.log()方法在控制台输出 Sum 名称，注意这里是在函数体的外部。

页面效果如图 8.5 所示。如图 8.5 中箭头所指，在函数体内部的 Sum 是一个 function 类型的对象，具体代表整个函数的代码，而函数体外部调用的 Sum 是一个未定义的名称。

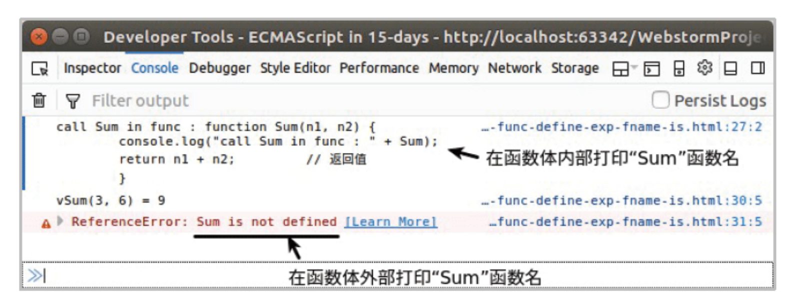


图 8.5　ECMAScript 函数表达式方式（3）

8.2.3　Function 构造方式定义 ECMAScript 函数

关于通过 Function 构造函数方式定义 ECMAScript 函数，具体语法格式如下：

```
var 变量名 = new Function("参数1", "参数2", ..., "参数n", "函数体");
```

其中，Function 构造函数可以接收任意数量的参数，最后一个参数为函数体，前面的参数枚举出新函数的参数。

下面是一个使用 Function 构造函数方式声明定义 ECMAScript 函数的代码示例（详见源代码 ch08 目录中的 ch08-es-func-define-Function.html 文件）。

【代码 8-6】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——Function 构造函数
04       */
05      var vSum = new Function("n1", "n2", "return n1+n2");
06      console.log("vSum(3, 6) = " + vSum(3, 6));
07  </script>
```

关于【代码 8-6】的分析如下：

第 05 行代码通过 var 关键字定义了一个函数变量 vSum，同时后面通过 new 操作符和 Function 关键字声明定义了函数，这就是 Function 构造函数方式声明定义 ECMAScript 函数的方法。

第 06 行代码直接通过函数名 vSum(3, 6)调用函数。

页面效果如图 8.6 所示。

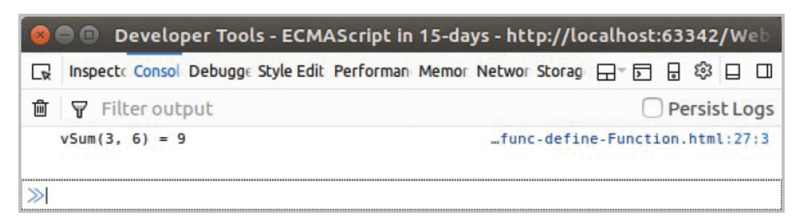


图 8.6　Function 构造函数方式

目前，通过 Function 构造函数方式声明定义 ECMAScript 函数的方法不是很常用，主要是因为该方式定义的函数没有被马上解释（需要到运行时才被解释），这样便导致了执行性能的降低。

8.3　ECMAScript 函数返回值

本节将介绍 ECMAScript 函数返回值的内容。在 ECMAScript 语法规范中，使用 return 关键字定义返回值。当然，也可以定义无返回值的函数，还可以仅使用 return 关键字而不带具体返回值的方式。

这里我们打算编写一个 ECMAScript 函数，实现计算两个整数的差值绝对值（正值）的操作。首先，看一个不使用返回值的 ECMAScript 函数如何实现这个功能（详见源代码 ch08 目录中的 ch08-es-func-return-none.html 文件）。

【代码 8-7】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——返回值
04       */
05      var v_dValue;
06      function dValue(n1, n2) {
07          if(n1 >= n2) {
08              v_dValue = n1 - n2;
09          } else {
10              v_dValue = n2 - n1;
11          }
12      }
13      dValue(3, 6);
14      console.log("dValue(3, 6) = " + v_dValue);
15  </script>
```

关于【代码 8-7】的分析如下：

第 05 行代码通过 var 关键字定义了一个全局变量 v_dValue，用来保存该差值。

第 06～12 行代码通过 function 关键字定义了一个函数 dValue。其中，第 07～11 行代码通过 if 条件语句计算两个整数的差值绝对值，并保存在全局变量 v_dValue 中，这里没有使用 return 关键字来返回值。

第 13 行代码通过函数名 dValue(3, 6)直接调用了该函数。这里之所以要单独调用一下该函数，是为了计算出差值并保存在全局变量 v_dValue 中。

第 14 行代码通过全局变量 v_dValue 在浏览器控制台中输出了差值结果。

页面效果如图 8.7 所示。差值（绝对值）计算的结果没有问题，但【代码 8-7】实现的过程似乎有点烦琐了。

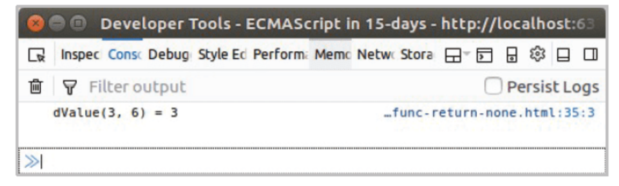


图 8.7　ECMAScript 函数返回值（无返回值方式）

下面看一下如何使用带返回值的 ECMAScript 函数实现该功能（详见源代码 ch08 目录中的 ch08-es-func-return-value.html 文件）。

【代码 8-8】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——返回值
04       */
05      function dValue(n1, n2) {
06          if(n1 >= n2) {
07              return n1 - n2;
08          } else {
09              return n2 - n1;
10          }
11      }
12      console.log("dValue(3, 6) = " + dValue(3, 6));
13  </script>
```

关于【代码 8-8】的分析如下：

第 05～11 行代码通过 function 关键字定义了一个函数 dValue。其中，第 06～10 行代码通过 if 条件语句计算两个整数的差值绝对值，并使用 return 关键字来返回值。

第 12 行代码通过函数名 dValue(3, 6)直接调用了该函数。

页面效果如图 8.8 所示。差值计算的结果同样正确输出，但对比【代码 8-7】和【代码 8-8】的实现过程，使用返回值的实现方式明显要简洁高效一些。

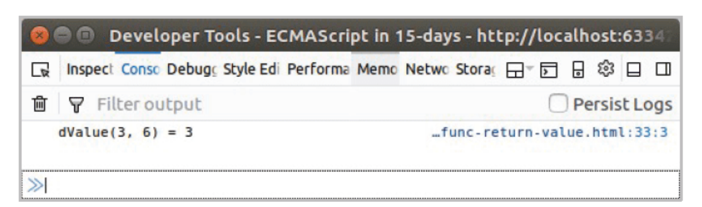


图 8.8　ECMAScript 函数返回值（带返回值方式）

最后，return 关键字还有一个比较常见的用法，就是仅使用 return 而不带具体返回值的方式可以中断函数的执行。

下面看一段单独使用 return 关键字的 ECMAScript 函数代码（详见源代码 ch08 目录中的 ch08-es-func-return-only.html 文件）。

【代码 8-9】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——返回值（单独使用）
04       */
05      function returnOnly(b) {
06          console.log("单独使用 return 可以中断函数的继续执行...");
07          if(b) {
08            console.log("函数执行过程将要被 return 中断...");
09              return;
10          }
11          console.log("函数执行完毕，看到效果了吧！ ^-^");
```

```
12        }
13        returnOnly(true);
14    </script>
```

关于【代码 8-9】的分析如下：

第 05～12 行代码通过 function 关键字定义了一个函数 returnOnly。其中，第 06 和第 11 行代码可以在浏览器控制台中各输出一行文字；第 07～10 行代码通过 if 条件选择语句判断函数 returnOnly()传递进来的参数是否要单独执行第 08 行代码，并执行第 09 行代码定义的 return 关键字。

第 13 行代码通过函数名 returnOnly(true)直接调用了该函数，这里的参数使用 true。

页面效果如图 8.9 所示。第 06 行和第 08 行代码分别在浏览器控制台中输出了一行文字，而第 11 行代码定义的文字却没有被输出。这是因为第 13 行代码调用函数 returnOnly(true)后，程序执行到第 09 行代码定义的 return 关键字时，提前中断了函数 returnOnly()的执行，从而导致第 11 行代码没有被执行。

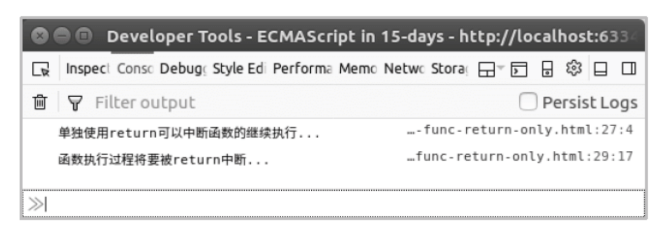


图 8.9　单独使用 return 关键字（1）

下面将【代码 8-9】中第 13 行代码所调用的函数 returnOnly(true)参数修改为 false（详见源代码 ch08 目录中的 ch08-es-func-return-only.html 文件）。

再次运行页面，查看控制台输出的调试信息，效果如图 8.10 所示。第 06 行和第 11 行代码分别在浏览器控制台中输出了一行文字，而第 08 行代码定义的文字却没有被输出。这是因为第 13 行代码调用函数 returnOnly(false)后，第 07～10 行代码通过 if 条件选择语句做出判断，第 09 行代码定义的 return 关键字没有被执行。因此，函数 returnOnly()自然也就没有被中断，第 11 行代码也就正常输出了。



图 8.10　单独使用 return 关键字（2）

8.4　arguments 对象

本节将介绍 ECMAScript 函数中 arguments 对象的内容。在 ECMAScript 语法规范中，使用特殊对象 arguments 不需要明确指出参数名也可以访问这些参数。对于 arguments 对象的设计，可以

增强 ECMAScript 函数使用的灵活性与功能性，如模拟实现函数重载功能等。

首先，看一段通过使用 arguments 对象的 length 属性来获取函数参数数量的代码（详见源代码 ch08 目录中的 ch08-es-func-arguments-length.html 文件）。

【代码 8-10】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——arguments 对象（length 属性）
04       */
05      function MessageBox() {
06          console.log("arguments.length = " + arguments.length);
07      }
08      MessageBox("Hello");    // 调用 ECMAScript 函数
09      MessageBox("Hello", "ECMAScript");    // 调用 ECMAScript 函数
10      MessageBox("Hello", "ECMAScript", "!");    // 调用 ECMAScript 函数
11  </script>
```

关于【代码 8-10】的分析如下：

第 05～07 行代码通过 function 关键字定义了一个函数 MessageBox()，注意该函数没有定义任何参数。其中，第 06 行代码通过 arguments 对象的 length 属性 arguments.length 获取了 arguments 对象的长度。

第 08～10 行代码分三次调用了 MessageBox()函数，且每次调用时的参数数量均不同。

运行页面，查看控制台输出的调试信息，效果如图 8.11 所示。arguments.length 属性值分别为 1、2 和 3，对应的就是三次调用函数 MessageBox()时使用的参数数量。

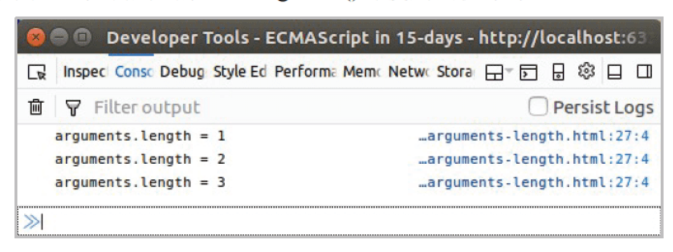


图 8.11　arguments 对象的 length 属性

上面这段代码介绍了 arguments.length 属性的使用方法。下面继续看一个结合使用 arguments.length 属性和 arguments 对象的代码示例（详见源代码 ch08 目录中的 ch08-es-func-arguments.html 文件）。

【代码 8-11】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——arguments 对象
04       */
05      function MessageBox() {
06          for(var i=0; i<arguments.length; i++) {
07              console.log("arguments[" + i + "] = " + arguments[i]);
08          }
09      }
10      MessageBox("1");    // 调用 ECMAScript 函数
```

```
11        MessageBox("a", "b");  // 调用 ECMAScript 函数
12        MessageBox("i", "ii", "iii");    // 调用 ECMAScript 函数
13    </script>
```

关于【代码 8-11】的分析如下：

第 05～09 行代码通过 function 关键字定义了一个函数 MessageBox()，注意该函数没有定义任何参数。

第 06～08 行代码通过 for 循环语句，对 arguments 对象的 length 属性 arguments.length 进行了迭代，用以获取 arguments 对象的每一次内容（通过 arguments[i]数组方式）。

第 08～10 行代码通过函数名 MessageBox()分三次调用了该函数，且每次调用时的参数数量及内容均不同。

运行页面，查看控制台输出的调试信息，效果如图 8.12 所示。三次调用函数 MessageBox()后，通过第 06～08 行代码的 for 循环语句，全部参数都输出在控制台中了。

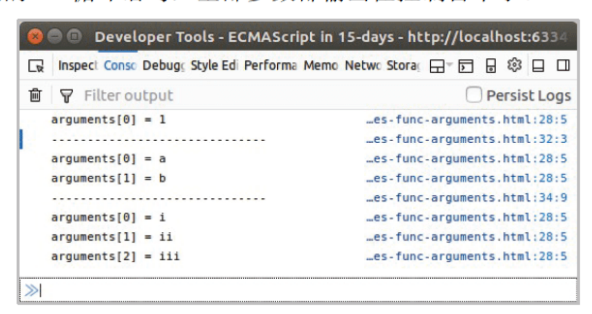


图 8.12　arguments 对象使用

实际上，【代码 8-11】已经十分接近对函数重载的使用方式了。

下面尝试通过 arguments 对象模拟一下 ECMAScript 函数重载的实现过程（详见源代码 ch08 目录中的 ch08-es-func-arguments-overload.html 文件）。

【代码 8-12】

```
01    <script type="text/javascript">
02        /*
03         * 定义 ECMAScript 函数——arguments 对象模拟函数重载
04         */
05        function Summary() {
06            var sum = 0;
07            var len = arguments.length;
08            for(var i=0; i<len; i++) {
09                sum += arguments[i];
10            }
11            return sum;
12        }
13        console.log("Summary(1) = " + Summary(1));     // 调用 ECMAScript 函数
14       console.log("Summary(1,2) = " + Summary(1, 2)); // 调用 ECMAScript 函数
15       console.log("Summary(1,2,3) = " + Summary(1, 2, 3)); //调用 ECMAScript 函数
16    </script>
```

关于【代码 8-12】的分析如下：

这段代码实现了一个累加器函数 Summary()，用于计算数字的累加和。既然是累加器，那运算

数的数量就不是固定的，这样使用传统函数方式实现起来就很麻烦，不过这正是函数重载的强项。

第 05～12 行代码通过 function 关键字定义了一个函数 Summary()，注意该函数没有定义任何参数。

第 06 行代码定义了第一个变量 sum，用于保存累加和。

第 07 行代码定义了第二个变量 len，用于保存 arguments.length 属性值。

第 08～10 行代码通过 for 循环语句对变量 len 进行了迭代，并使用加法/赋值运算符（+=）实现了数字累加计算。

第 11 行代码通过 return 关键字返回计算得出的累加和。

第 13～15 行代码通过函数名 Summary()分三次调用该函数进行了测试，且每次调用时的参数数量及数值均不同。

控制台输出的调试信息如图 8.13 所示。三次调用累加器函数 Summary()后，每次累加和的结果全部输出在控制台中了。

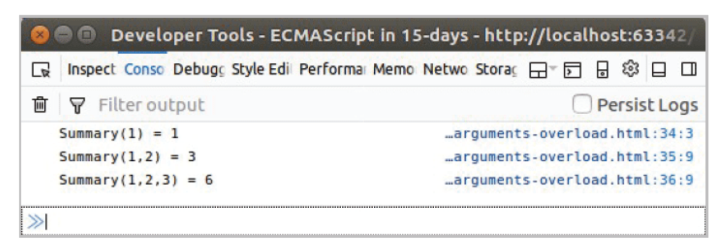


图 8.13　arguments 对象模拟函数重载

8.5　Function 对象

本节将介绍 ECMAScript 函数中 Function 对象的内容。前文中介绍过 Function 构造函数的方式，这是源自于在 ECMAScript 语法规范中 Function 被定义为对象的缘故（一切皆为对象）。在 ECMAScript 语法规范中，Function 对象可用于定义任何函数，在这一点上很像 C++和 Java 语言中“类”的概念。

下面通过几个关于 Function 对象的具体应用实例，详细介绍一下如何使用 Function 对象。

8.5.1　Function 对象实现函数指针

首先来看一个通过使用 Function 对象实现函数指针的代码示例（详见源代码 ch08 目录中的 ch08-es-func-Function-pointer.html 文件）。

【代码 8-13】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——Function 对象（函数指针）
04       */
05      var vSum = new Function("n1", "n2", "return n1+n2");
```

```
06        console.log("vSum(3, 6) = " + vSum(3, 6));
07        var vSumPointer = vSum;
08        console.log("vSumPointer(3, 6) = " + vSumPointer(3, 6));
09    </script>
```

关于【代码 8-13】的分析如下：

这段代码实际上是在【代码 8-6】的基础上进行改写的，目的是实现一个函数指针的应用实例。

第 05 行代码通过 var 关键字定义了第一个函数变量 vSum，同时后面通过 new 操作符和 Function 关键字声明定义了函数。

第 06 行代码直接通过函数名 vSum(3, 6)调用函数。

第 07 行代码通过 var 关键字定义了第二个变量 vSumPointer，并指向了第一个函数变量 vSum，这其实就是一个函数指针。

第 08 行代码直接通过变量名 vSumPointer(3, 6)调用函数。

运行页面，控制台输出的调试信息如图 8.14 所示。函数指针 vSumPointer 与原函数 vSum 实现了同样的功能，是可以相互替换的。

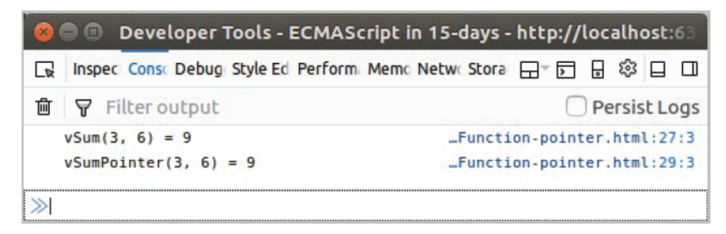


图 8.14　Function 对象实现函数指针

现在是不是有点意思了呢？既然函数指针可以通过定义指向原函数的变量来实现，自然也可以作为参数来传递了。

下面继续看一个将函数指针作为参数来传递的代码示例（详见源代码 ch08 目录中的 ch08-es-func-Function-pointer-param.html 文件）。

【代码 8-14】

```
01    <script type="text/javascript">
02        /*
03         * 定义 ECMAScript 函数——Function 对象（函数指针作为参数）
04         */
05        var vSum = new Function("n1", "n2", "return n1+n2");
06        console.log("vSum(3, 6) = " + vSum(3, 6));
07        function vSumSum(fn, n1, n2) {
08            return fn(n1, n2);
09        }
10        console.log("vSumSum(vSum, 3, 6) = " + vSumSum(vSum, 3, 6));
11    </script>
```

关于【代码 8-14】的分析如下：

这段代码同样也是在【代码 8-6】的基础上进行改写的，目的是实现一个函数指针作为参数传递的应用实例。

第 05 行代码通过 var 关键字定义了第一个函数变量 vSum，同时后面通过 new 操作符和 Function 关键字声明定义了函数。

第 06 行代码直接通过函数名 vSum(3, 6)调用函数。

第 07～09 行代码定义了一个函数 vSumSum()，同时定义了 3 个参数，注意第一个参数其实是当作函数指针来使用的。其中，第 08 行代码通过 return 关键字返回了一个函数 fn(n1, n2)。

第 10 行代码直接通过函数名 vSumSum(vSum, 3, 6)调用函数。

运行页面，控制台输出的调试信息如图 8.15 所示。通过第 10 行代码的输出结果来看，第 07～09 行代码定义的函数 vSumSum()，将第一个参数作为函数指针来传递是成功的。另外，第 06 行与第 10 行的输出结果也是完全一致的。

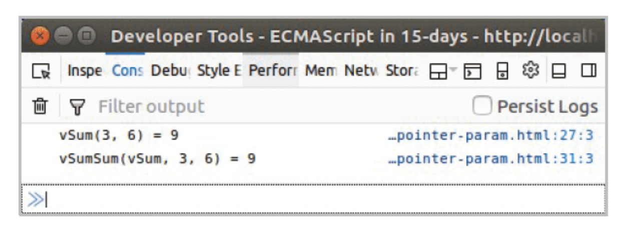


图 8.15　Function 对象实现函数指针作为参数

8.5.2　Function 对象属性

现在，既然明确了在 ECMAScript 语法规范中 Function 是作为对象来定义的，那么 Function 对象自然也会有属性。在 ECMAScript 语法规范中，Function 对象可以使用共有的 length 属性来表示参数的数量，这一点与 arguments 对象是一致的。

下面是一段使用 Function 对象中 length 属性的代码（详见源代码 ch08 目录中的 ch08-es-func-Function-length.html 文件）。

【代码 8-15】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——Function 对象（length 属性）
04       */
05      function MessageBox() {
06      }
07      console.log("MessageBox.length="+MessageBox.length);调用 ECMAScript 函数
08      function Sum(n1, n2) {
09          return n1 + n2;
10      }
11      console.log("Sum.length = " + Sum.length);     // 调用 ECMAScript 函数
12  </script>
```

关于【代码 8-15】的分析如下：

第 05 和第 06 行代码通过 function 关键字定义了第一个函数 MessageBox(),注意该函数没有定义任何参数。

第 07 行代码通过使用函数名 MessageBox 的 length 属性 MessageBox.length 获取了函数 MessageBox()所定义参数的数量。

第 08～10 行代码通过 function 关键字定义了第二个函数 Sum(n1, n2)，注意该函数定义了两个参数。

第 11 行代码通过使用函数名 Sum 的 length 属性 Sum.length 获取了函数 Sum()所定义参数的数量。

运行页面，控制台输出的调试信息如图 8.16 所示。通过第 07 行代码的输出结果来看，函数 MessageBox()定义参数的数量为 0。通过第 11 行代码的输出结果来看，函数 Sum(n1, n2)定义参数的数量为 2，输出的结果与实际定义情况是完全吻合的。

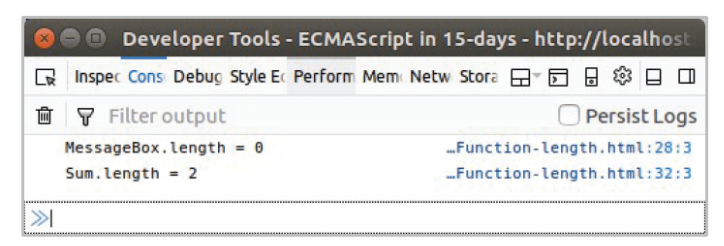


图 8.16　Function 对象的 length 属性

8.5.3　Function 对象方法

既然 Function 对象支持共有属性，自然也会支持共有方法。在 ECMAScript 语法规范中，Function 对象可以使用共有的 toString()方法来实现函数源代码的输出。

下面是一段使用 Function 对象中 toString()方法的代码（详见源代码 ch08 目录中的 ch08-es-func-Function-toString.html 文件）。

【代码 8-16】

```
01  <script type="text/javascript">
02      /*
03       * 定义 ECMAScript 函数——Function 对象（toString()方法）
04       */
05      function MessageBox() {
06      }
07      console.log("MessageBox.toString() = " + MessageBox.toString());
08      function Sum(n1, n2) {
09          return n1 + n2;
10      }
11      console.log("Sum.toString() = " + Sum.toString()); // 调用 ECMAScript 函数
12  </script>
```

关于【代码 8-16】的分析如下：

这段代码是在【代码 8-15】的基础上改写的，把对 length 属性的使用改成了对 toString()方法的使用。

第 07 行代码通过使用函数名 MessageBox 的 toString()方法 MessageBox.toString()输出了函数 MessageBox()的源代码。

第 11 行代码通过使用函数名 Sum 的 toString()方法 Sum.toString()输出了函数 Sum()的源代码。

运行页面，控制台输出的调试信息如图 8.17 所示。通过使用函数对象的 toString()方法，成功输出了函数定义的源代码，这个功能在代码的调试开发中是非常有用的。

```
MessageBox.toString() = function                 …Function-toString.html:28:2
MessageBox() {
        }
------------------------------                   …Function-toString.html:29:2
Sum.toString() = function Sum(n1, n2) {          …Function-toString.html:33:2
                return n1 + n2;
        }
```

图 8.17　Function 对象的 toString()方法

8.6　本章小结

本章主要介绍了 ECMAScript 语法规范中关于函数的知识，包括 ECMAScript 函数的声明、定义与调用，ECMAScript 函数返回值，ECMAScript 函数中 arguments 对象和 Function 对象等方面的内容。

第 9 章

系统函数

本章将介绍 ECMAScript 语法规范中关于系统函数的内容。ECMAScript 脚本语言为设计人员提供了大量的系统函数（内置函数），这些函数无须设计人员声明或引用，可以直接进行使用（由浏览器提供支持）。ECMAScript 系统函数包括常规函数、字符串函数、数学函数、数组函数和日期函数五大类。

9.1 ECMAScript 常规函数

ECMAScript 常规函数包括一些常见的弹出对话框、类型转换、执行代码和判断数字函数，下面详细进行介绍。

9.1.1 常规函数介绍

ECMAScript 语法规范中定义的常规函数主要包括以下几种：

- alert 函数：显示一个警告对话框，包括一个 OK 按钮。
- confirm 函数：显示一个确认对话框，包括 OK 和 Cancel 按钮。
- parseInt 函数：将字符串转换为整数形式（可指定几进制）。
- isNaN 函数：判断是否为非数字。
- eval 函数：计算字符串的结果，执行 JavaScript 脚本代码（注意参数仅接受原始字符串）。

9.1.2 警告对话框和确认对话框

ECMAScript 语法规范中定义的警告对话框（alert）和确认对话框（confirm）是常用的系统函数。

下面是一个同时使用警告对话框（alert）和确认对话框（confirm）的代码示例（详见源代码 ch09 目录中的 ch09-es-alert-confirm.html 文件）。

【代码 9-1】

```
01  <script type="text/javascript">
02      var bConfirm = confirm("请选择确认!");
03      if(bConfirm) {
04          alert("ok");
05      } else {
06          alert("cancel");
07      }
08  </script>
```

关于【代码 9-1】的分析如下：

第 02 行代码通过 confirm()函数定义了一个确认对话框，并将返回值赋给变量 bConfirm，此时该变量为一个布尔类型。

第 03～07 行代码通过判断变量 bConfirm 的布尔值是“真”或“假”来选择弹出不同内容的警告对话框（alert()）。

第 04 行和第 06 行代码通过 alert()函数定义了两个警告对话框。

运行页面，初始效果如图 9.1 所示。页面中弹出的是第 02 行代码定义的确认对话框。若单击 OK 按钮，则页面执行后的效果如图 9.2 所示。

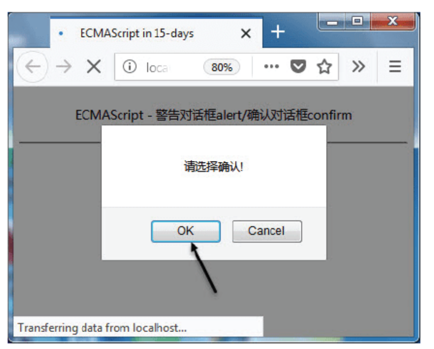


图 9.1　ECMAScript 警告对话框和确认对话框（1）

图 9.2　ECMAScript 警告对话框和确认对话框（2）

若尝试在图 9.1 中单击 Cancel 按钮，则页面执行后的效果如图 9.3 所示。

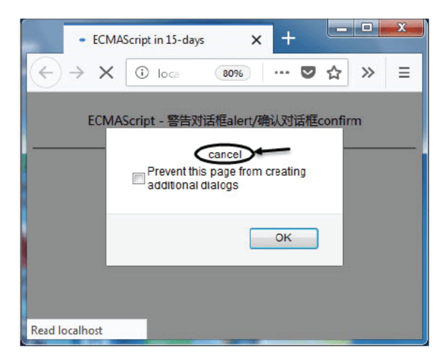


图 9.3　ECMAScript 警告对话框和确认对话框（3）

9.1.3　parseInt()函数

在 ECMAScript 语法规范中定义的 parseInt()函数用于解析一个字符串，然后返回一个整数。

关于 parseInt()函数的语法格式如下：

```
parseInt(string, radix);
```

其中，参数 string 用于表示要被解析的字符串数值。若该值以“0x”开头，则以十六进制为基数。

参数 radix 是可选的，用于表示要解析字符串数值的基数。若省略 radix 参数或其值为 0，则以十进制为基数进行解析；若其值为 8 或 16，则以八进制或十六进制为基数进行解析。

下面是一个使用 parseInt()函数进行解析整数的代码示例（详见源代码 ch09 目录中的 ch09-es-parseInt.html 文件）。

【代码 9-2】

```
01  <script type="text/javascript">
02       "use strict";
03       console.log("parseInt('10') = " + parseInt('10'));
04      console.log("parseInt('10', 10) = " + parseInt('10', 10));
05      console.log("parseInt('10', 2) = " + parseInt('10', 2));
06      console.log("parseInt('77', 8) = " + parseInt('77', 8));
07      console.log("parseInt('ff', 16) = " + parseInt('ff', 16));
08      console.log("parseInt('0o10') = " + parseInt('0o10'));
09      console.log("parseInt('0x10') = " + parseInt('0x10'));
10  </script>
```

关于【代码 9-2】的分析如下：

第 03 行代码通过 parseInt('10')函数来解析字符串'10'。注意，这里没有使用 radix 参数，表示默认转换为十进制整数。

第 04 行代码再次通过 parseInt('10')函数来解析字符串'10'，与第 03 行代码不同的是定义了 radix 参数 10，表示转换为十进制整数。

第 05 行代码第三次通过 parseInt('10')函数来解析字符串'10'，但这里定义了 radix 参数 2，表示转换为二进制整数。

第 06 行代码通过 parseInt('77')函数解析了字符串'77'，同时定义了 radix 参数 8，表示转换为八进制整数。

第 07 行代码通过 parseInt('ff')函数解析了字符串'ff'，同时定义了 radix 参数 16，表示转换为十六进制整数。

第 08 行代码通过 parseInt('0o10')函数解析了字符串'0o10'，这里定义的字符串 0o10 其实是八进制表示法。

第 09 行代码通过 parseInt('0x10')函数解析了字符串'0x10'，这里定义的字符串 0x10 其实是十六进制表示法。

运行页面，查看控制台输出的调试信息，如图 9.4 所示。

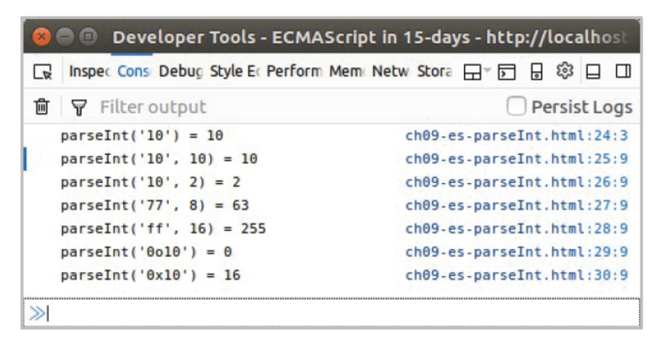


图 9.4　parseInt()函数方法

如图 9.4 所示，通过对比第 03 行和第 04 行代码输出的结果，可以看到 radix 参数默认值就是 10，因此如果打算转换为十进制整数的话，那么定义或不定义 radix 参数都是可以的。

从第 05～07 行代码输出的结果来看，parseInt()函数将会根据 radix 参数定义的进制值转换为相应的进制整数。

通过对比第 08 行和第 09 行代码输出的结果，可以看到 parseInt()函数是无法识别八进制字符串'0o10'的，但是却可以识别十六进制字符串'0x10'，这一点需要读者注意。

9.1.4　isNaN()函数

在 ECMAScript 语法规范中定义了一个用于判断某个值是否为非数字的函数方法——isNaN()。其实，“NaN”是一个英文缩写，全拼是“Not a Number”。

下面是一个使用 isNaN()函数判断是否为非数字的代码示例（详见源代码 ch09 目录中的 ch09-es-isNaN.html 文件）。

【代码 9-3】

```
01  <script type="text/javascript">
02   if(isNaN(NaN)) {
03        console.log("NaN return true.");
04   } else {
05        console.log("NaN return false.");
06   }
07   if(isNaN(null)) {
08        console.log("null return true.");
09   } else {
10        console.log("null return false.");
11   }
12   if(isNaN(undefined)) {
13        console.log("undefined return true.");
14   } else {
15        console.log("undefined return false.");
16   }
17   if(isNaN(true)) {
18        console.log("true return true.");
19   } else {
20        console.log("true return false.");
21   }
```

```
22  if(isNaN(123)) {
23      console.log("123 return true.");
24  } else {
25      console.log("123 return false.");
26  }
27  if(isNaN("123")) {
28      console.log("'123' return true.");
29  } else {
30      console.log("'123' return false.");
31  }
32  if(isNaN("abc")) {
33      console.log("'abc' return true.");
34  } else {
35      console.log("'abc' return false.");
36  }
37  if(isNaN("")) {
38      console.log("'' return true.");
39  } else {
40      console.log("'' return false.");
41  }
42 </script>
```

关于【代码 9-3】的分析如下：

这段代码主要通过 isNaN()函数判断一些 ECMAScript（JavaScript）保留字、数字和字符串是否为非数字。

运行页面，控制台输出的调试信息如图 9.5 所示。通过 isNaN()函数判断保留字 NaN、undefined 均为非数字，字符串 "abc" 也是非数字。isNaN()函数判断 null、true 和空字符串均为数字，判断 123 和 "123" 也是数字。根据 ECMAScript 语法规范中的定义，因为 isNaN()函数是将 null、true 和空字符串当作 0 来处理的，所以会返回数字。

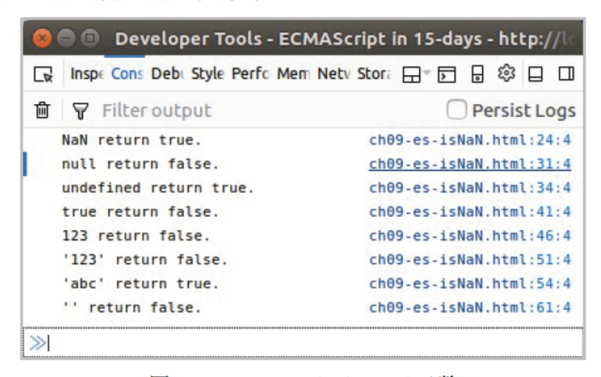


图 9.5　ECMAScript isNaN()函数

9.1.5　eval()函数

在 ECMAScript 语法规范中有一个非常实用的 eval()函数方法，用于执行 JavaScript 脚本代码。下面就是一个使用 eval()函数的代码示例（详见源代码 ch09 目录中的 ch09-es-eval.html 文件）。

【代码 9-4】

```
01  <script type="text/javascript">
02      console.log("print : 1 + 2 = 3");
03      var x = 1;
04      console.log("x = " + x);
05      var y = 2;
06    console.log("y = " + y);
07      console.log("x + y = " + (x + y));
08    console.log("print by eval() : 1 + 2 = 3");
09      eval("x=1;console.log('x = ' + eval(x));");
10      eval("y=2;console.log('y = ' + eval(y));");
11      eval("console.log('x + y = ' + eval(x+y));");
12  </script>
```

关于【代码 9-4】的分析如下：

这段代码主要是模拟输出算式（1+2=3）的计算过程。其中，第 03～07 行代码通过传统方式进行输出；第 09～11 行代码通过 eval()函数进行输出，读者可以对比一下这两种方式。

第 09 行代码其实是通过 eval()函数执行了一段 JavaScript 代码，具体就是第 03 和第 04 行代码中定义、初始化和输出变量 x 的过程。

第 10 行代码与第 09 行代码一样，通过 eval()函数执行了第 05 和第 06 行代码的定义、初始化和输出变量 y 的过程。

第 11 行代码再次通过 eval()函数执行了一段 JavaScript 代码，具体就是第 07 行代码中变量 x 与变量 y 相加的运算。

运行页面，控制台输出的调试信息如图 9.6 所示。通过使用 eval()函数可以很好地实现解析 JavaScript 脚本代码的功能。需要特别注意，eval()函数内的参数是一段字符串，解析执行的也是字符串。这是因为根据 ECMAScript 语法规范中要求，eval()函数内的参数必须是字符串，不论参数中包含多少条要执行的语句。

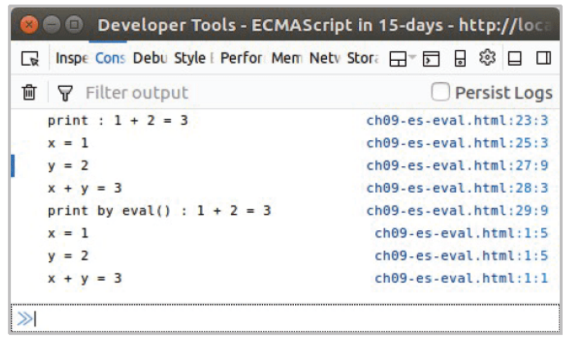


图 9.6　ECMAScript eval()函数

9.2　ECMAScript 字符串函数

ECMAScript 字符串函数包括一些常见的索引字符、定位下标、获取长度和大小写转换的函数方法，下面进行详细介绍。

ECMAScript 语法规范中定义的字符串函数主要包括以下几种：

（1）charAt 函数：返回字符串中指定的某个字符。

charAt 函数的语法如下：

```
stringObject.charAt(index)          //可返回指定位置的字符
```

其中，index 参数是必需的，表示字符在字符串中的下标数值。

（2）indexOf 函数：返回某个指定字符在字符串中首次出现的下标位置，从字符串左边开始查找。

indexOf 函数的语法如下：

```
stringObject.indexOf(searchvalue,fromindex) // 返回某个指定的字符串值在字符串中首次出现的下标位置
```

其中，searchvalue 参数是必需的，表示需要检索的字符串值。

fromindex 参数是可选的整数值，规定在字符串中开始检索的位置。其合法取值是 0 到 stringObject.length-1。若省略该参数，则将从字符串的首字符开始检索。

（3）lastIndexOf 函数：返回字符串中某个指定字符在字符串中首次出现的下标位置，从字符串右边开始查找。

lastIndexOf 函数的语法如下：

```
stringObject.lastIndexOf(searchvalue,fromindex)     // 返回一个指定的字符串值最后出现的位置，在一个字符串中的指定位置从后向前搜索
```

其中，searchvalue 参数是必需的，表示需要检索的字符串值。

fromindex 参数是可选的整数值，规定在字符串中开始检索的位置。其合法取值是 0 到 stringObject.length-1。若省略该参数，则将从字符串的最后一个字符开始检索。

（4）length 函数：返回字符串的长度。

（5）substring 函数：返回字符串中指定的几个字符（参数非负）。

substring 函数的语法如下：

```
stringObject.substring(start,stop)     // 用于提取字符串中介于两个指定下标之间的字符
```

其中，start 参数是必需的，为一个非负的整数，规定要提取子串的开始字符在 stringObject 中的位置。

stop 参数是可选的，表示一个非负的整数，比要提取子串的最后一个字符在 stringObject 中的位置多 1。若省略该参数，则返回的子串一直到字符串的结尾。

注意：若定义了 stop 参数，则返回的子串包括 start 处的字符，但不包括 stop 处的字符。

（6）substr 函数：返回字符串中指定的几个字符（参数可负）。

substr 函数的语法如下：

```
stringObject.substr(start,length)          // 在字符串中抽取从 start 下标开始的指定长度的字符串
```

其中，start 参数是必需的，定义抽取子串的起始下标，必须是数值类型；如果是负数，那么该参数声明从字符串的尾部开始算起始的位置。例如，-1 指字符串中最后一个字符，-2 指倒数第二个字符，以此类推。

length 参数是可选的，定义子串的字符数必须是数值类型。如果省略了该参数，那么返回从 start

开始位置到结尾的字符串。

注意：由于 ECMAScript 标准中未对该方法进行标准化，因此不建议使用该函数。

（7）toLowerCase 函数：将字符串中字母转换为小写。

（8）toUpperCase 函数：将字符串中字母转换为大写。

下面是一个综合使用以上字符串函数的代码示例（详见源代码 ch09 目录中的 ch09-es-string.html 文件）。

【代码 9-5】

```
01  <script type="text/javascript">
02      var str = "abcdefghijklmnopqrstuvwxyz";
03      console.log(str.charAt(0));
04      console.log(str.indexOf("c"));
05      console.log(str.lastIndexOf("c"));
06      console.log(str.length);
07      console.log(str.substring(18));
08      console.log(str.substring(8, 16));
09      console.log(str.toUpperCase());
10      console.log(str.toLowerCase(str.toUpperCase()));
11  </script>
```

关于【代码 9-5】的分析如下：

第 02 行代码定义了一个字符串变量 str，并初始化了 26 个英文小写字母。

第 03 行代码通过 charAt()函数获取了字符串变量 str 的第一个字符，注意下标数值为 0。

第 04 行代码通过 indexOf()函数获取了字符串变量 str 中字符 "c" 的下标数值。

第 05 行代码通过 lastIndexOf()函数同样是获取了字符串变量 str 中字符 "c" 的下标数值。

第 06 行代码通过 length 函数方法获取了字符串变量 str 的字符长度。

第 07 行代码通过 substring()函数获取了字符串变量 str 中从下标数值 18 开始到字符串结束的子字符串。

第 08 行代码通过 substring()函数获取了字符串变量 str 中从下标数值 8 开始到下标数值 16 的子字符串。

第 09 行代码通过 toUpperCase()函数将字符串变量 str 全部转换为大写字母。

第 10 行代码在第 09 行代码的基础上，通过 toLowerCase()函数将字符串变量 str 再次转换为小写字母。

运行页面，控制台输出的调试信息如图 9.7 所示。

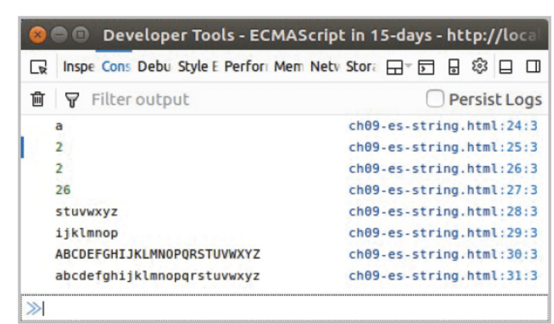


图 9.7　ECMAScript 字符串函数

9.3　ECMAScript 数学函数

ECMAScript 数学函数包括一些常见的绝对值、三角函数、最大最小整数和随机数的计算函数，下面进行详细介绍。

ECMAScript 语法规范中定义的数学函数主要是通过 Math 对象来实现的，具体包括以下函数（按照字母顺序排列）：

- abs 函数：返回一个数字的绝对值。
- acos 函数：返回一个数字的反余弦值，结果为 0～π 的弧度。
- asin 函数：返回一个数字的反正弦值，结果为-π/2～π/2 的弧度。
- atan 函数：返回一个数字的反正切值，结果为-π/2～π/2 的弧度。
- ceil 函数：返回一个数字的最小整数值（大于或等于）。
- cos 函数：返回一个数字的余弦值，结果为-1～1。
- exp 函数：返回 e（自然对数）的乘方值。
- floor 函数：返回一个数字的最大整数值（小于或等于）。
- log 函数：自然对数函数，返回一个数字的自然对数（e）值。
- max 函数：返回两个数的最大值。
- min 函数：返回两个数的最小值。
- pow 函数：返回一个数字的乘方值。
- random 函数：返回一个 0～1 的随机数值。
- round 函数：返回一个数字的四舍五入值，类型是整数。
- sin 函数：返回一个数字的正弦值，结果为-1～1。
- sqrt 函数：返回一个数字的平方根值。
- tan 函数：返回一个数字的正切值。

以上这些数学函数在纯数学计算方面的编程中是非常有用的。下面是一个使用部分数学函数的代码示例（详见源代码 ch09 目录中的 ch09-es-math.html 文件）。

【代码 9-6】

```
01  <script type="text/javascript">
02      console.log("Math.max(2, 3) = " + Math.max(2, 3));
03      console.log("Math.min(2, 3) = " + Math.min(2, 3));
04      console.log("Math.ceil(2 / 3) = " + Math.ceil(2 / 3));
05      console.log("Math.floor(2 / 3) = " + Math.floor(2 / 3));
06      console.log("Math.round(2 / 3) = " + Math.round(2 / 3));
07      console.log("Math.pow(2, 3) = " + Math.pow(2, 3));
08      console.log("Math.random() : " + Math.random());
09    console.log("计算产生 0～100 随机数 : " + (100 * Math.random()));
10    console.log("计算产生 0～100 随机整数 : " + Math.round(100 * Math.random()));
11  </script>
```

关于【代码 9-6】的分析如下：

第 02～10 行代码分别通过一系列数学函数进行了相应的运算，并在控制台输出了调试信息。

运行 HTML 页面，控制台输出的调试信息如图 9.8 所示。

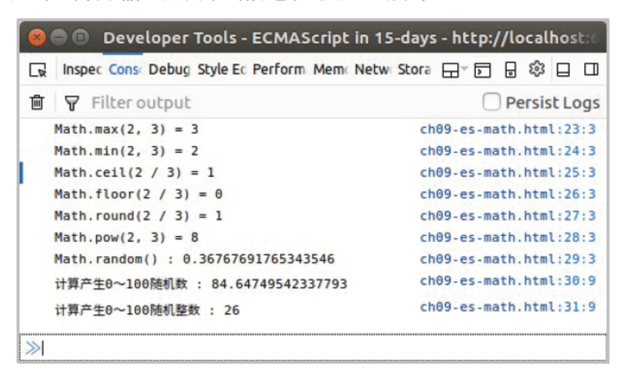


图 9.8　ECMAScript 数学函数（Math）

9.4　ECMAScript 数组函数

ECMAScript 数组函数包括一些常见针对数组的连接、倒序和排序的函数方法，下面进行详细介绍。

9.4.1　数组函数介绍

ECMAScript 语法规范中定义关于数组的函数主要包括以下几种：

（1）join 函数：转换并连接数组中的所有元素为一个字符串。

join 函数的语法如下：

```
arrayObject.join(separator)
```

其中，separator 参数为可选的，表示要使用的分隔符。若省略该参数，则使用逗号作为分隔符。

（2）reverse 函数：将数组元素顺序颠倒。

reverse 函数的语法如下：

```
arrayObject.reverse()
```

注　意
该函数仅仅改变原来的数组，不会创建新数组。

（3）sort 函数：将数组元素重新排序。

sort 函数的语法如下：

```
arrayObject.sort(sortby)
```

其中，sortby 参数是可选的，用于规定排列顺序。若使用该参数，则必须是一个比较函数。

（4）from 函数：将类似数组的对象或可迭代的对象转换为真正的数组。

from 函数的语法如下：

```
Array.from(arr)
```

其中，arr 参数定义为一个类似数组或可迭代的对象。

（5）length 函数：返回数组的长度。

9.4.2　join 函数

在 ECMAScript 数组函数中，join()函数用于连接数组中的所有元素并转换为一个字符串。下面是一个数组连接（join 函数）的代码示例（详见源代码 ch09 目录中的 ch09-es-arr-join.html 文件）。

【代码 9-7】

```
01  <script type="text/javascript">
02   var arr = new Array('a', 'b', 'c');
03   console.log(arr);
04   console.log(arr.join());
05   console.log(arr.join('-'));
06  </script>
```

关于【代码 9-7】的分析如下：

第 02 行代码定义了一个数组变量 arr，并进行了初始化操作。

第 03 行代码直接在控制台输出了数组变量 arr 的调试信息。

第 04 行代码对数组变量 arr 使用了 join()函数，并在控制台输出了该数组变量 arr 的调试信息。

第 05 行代码对数组变量 arr 使用了设定分隔符"-"为 join()函数的参数，并在控制台输出了该数组变量 arr 的调试信息。

运行页面，控制台输出的调试信息如图 9.9 所示。第 04 行代码使用不带参数的 join()函数会默认添加逗号"，"作为分隔符。

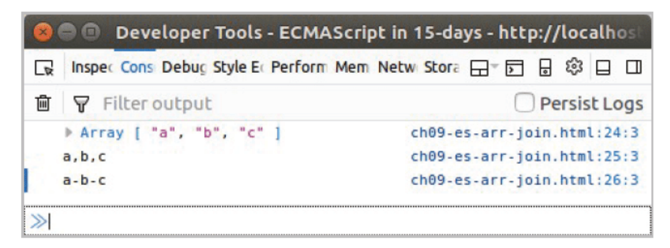


图 9.9　ECMAScript 数组函数 join()

9.4.3　reverse 函数

在 ECMAScript 数组函数中，reverse()函数用于将数组项顺序颠倒。下面是一个数组顺序颠倒（reverse 函数）的代码示例（详见源代码 ch09 目录中的 ch09-es-arr-reverse.html 文件）。

【代码 9-8】

```
01  <script type="text/javascript">
02   var arr = new Array('a', 'b', 'c');
03   console.log(arr);
04   console.log(arr.reverse());
05  </script>
```

关于【代码 9-8】的分析如下：

第 02 行代码定义了一个数组变量 arr，并进行了初始化操作。

第 03 行代码直接在控制台输出了数组变量 arr 的调试信息。

第 04 行代码对数组变量 arr 使用 reverse()函数进行了顺序颠倒操作，并在控制台输出了该数组变量 arr 的调试信息。

运行页面，控制台输出的调试信息如图 9.10 所示。

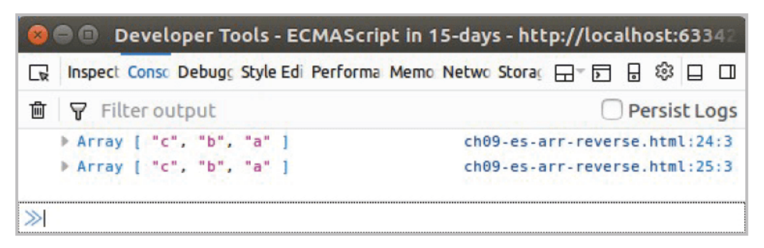


图 9.10　ECMAScript 数组函数 reverse()

9.4.4　sort 函数

在 ECMAScript 数组函数中，sort()函数用于将数组元素重新排序。下面是一个进行数组排序（sort 函数）和计算数组长度（length 属性）的代码示例（详见源代码 ch09 目录中的 ch09-es-arr-sort.html 文件）。

【代码 9-9】

```
01   <script type="text/javascript">
02        var arr_str = new Array('9', '6', '15', '3', '333', '88');
03        console.log(arr_str.length);
04        console.log(arr_str.sort());
05        var arr_num = new Array(9, 6, 15, 3, 333, 88);
06        console.log(arr_num.length);
07        console.log(arr_num.sort());
08        /*
09         * 定义用于 sort 函数进行排序的方法
10         */
11        function sortBy(a, b) {
12            return a - b;
13        }
14        console.log(arr_num.sort(sortBy));
15   </script>
```

关于【代码 9-9】的分析如下：

第 02 行代码定义了第一个字符数组变量 arr_str，并进行了初始化操作。

第 03 行代码对数组变量 arr_str 使用 length 函数方法获取数组长度，并在控制台输出调试信息。

第 04 行代码对数组变量 arr_str 使用 sort()函数进行排序操作，并在控制台输出数组变量 arr_str 的调试信息。

第 05 行代码定义了第二个整数型数组变量 arr_num，并进行了初始化操作。

第 06 行代码对数组变量 arr_num 使用 length 函数方法获取数组长度，并在控制台输出调试信息。

第 07 行代码对数组变量 arr_num 使用 sort()函数进行排序操作，并在控制台输出数组变量 arr_num 的调试信息。

第 11～13 行代码定义了一个用于排序的函数 sortBy，该函数将作为 sort()函数的参数来使用，后面我们会详细介绍 sortBy 函数。

第 14 行代码对数组变量 arr_num 使用 sort()函数进行排序操作。注意，增加了排序参数 sortBy，再次在控制台输出数组变量 arr_num 的调试信息。

运行页面，控制台输出的调试信息如图 9.11 所示。

如图 9.11 中箭头所指，无论是对于字符数组或整数型数组，使用 sort()函数排序时都会将数组项视为字符来进行排序，这一点从第 04 和第 07 行代码输出的结果就可以判断出。

因此，若想让 sort()函数对整数型数组进行正常排序，则需要定义排序函数 sortBy 作为 sort()函数的参数。其中，排序函数 sortBy 中的 a、b 参数用于定义排序规则，a 表示前一个数组项，b 表示后一个数组项；“a-b”的结果若小于 0 则表示 a 小于 b，数组项顺序不变；“a-b”的结果若大于 0 则表示 a 大于 b，数组项顺序颠倒；“a-b”的结果若等于 0 则表示 a 等于 b，数组项不进行排序。所以，第 14 行代码使用了带 sortBy 参数的 sort()函数后，才能得到整数型数组 arr_num 的正确排序（由小到大）。

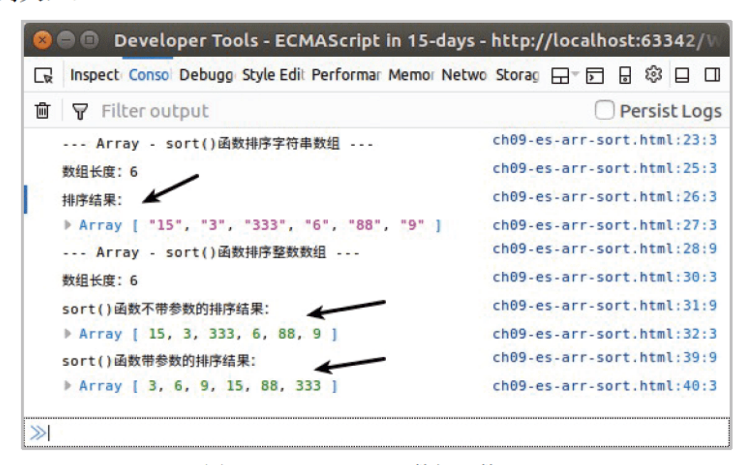


图 9.11　ECMAScript 数组函数 sort()

9.4.5　from 函数

在 ECMAScript 数组函数中，from()函数用于将一个类似数组的对象转换为真正的数组。另外，Array 对象的 from()函数是 ECMAScript 6 版本中新增的一个函数方法。

下面是一个使用数组转换（from 函数）的代码示例（详见源代码 ch09 目录中的

ch09-es-arr-from.html 文件）。

【代码 9-10】

```
01  <script type="text/javascript">
02       /**
03        * 将类似数组对象转换为数组
04        */
05       var arrLike = {
06           0: 'a',
07             1: 'b',
08             2: 'c',
09             length: 3
10       };
11       let arrL = Array.from(arrLike);
12       console.log(arrL);
13      /**
14       * 将可迭代对象转换为数组
15       */
16       var str = "ECMAScript";
17     let arrC = Array.from(str);
18     console.log(arrC);
19     /**
20      * 将函数内部对象（arguments）转换为数组
21      */
22     function func(a, b, c) {
23         let args = Array.from(arguments);
24         console.log(args);
25       }
26       func(1, 2, 3);
27  </script>
```

关于【代码 9-10】的分析如下：

第 05～10 行代码定义了一个类似数组的对象 arrLike，第 11 行代码通过 from()函数将该对象转换为数组。

第 16 行代码定义了一个字符串（可迭代的），第 17 行代码通过 from()函数将该字符串转换为数组。

第 22～25 行代码定义了一个函数 func，第 23 行代码通过 from()函数将该函数 func 内部的对象 arguments 转换为数组。

运行页面，控制台输出的调试信息如图 9.12 所示。

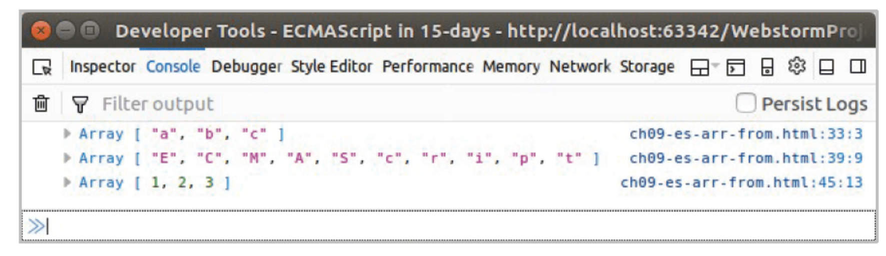


图 9.12　ECMAScript 数组函数 from()

9.5　ECMAScript 日期函数

ECMAScript 日期函数包括一些常见的针对年、月、日、星期、小时、分钟和秒数的函数方法，下面进行详细介绍。

ECMAScript 语法规范中定义了一组日期函数，比较常用的有以下几种：

- getYear 函数：返回日期的“年”部分，返回值以 1900 年为基数。
- getMonth 函数：返回日期的“月”部分，值为 0～11。
- getDay 函数：返回星期几，值为 0～6。其中，0 表示星期日、1 表示星期一、...、6 表示星期六。
- getDate 函数：返回日期的“日”部分，值为 1～31。
- getHours 函数：返回日期的“小时”部分，值为 0～23。
- getMinutes 函数：返回日期的“分钟”部分，值为 0～59。
- getSeconds 函数：返回日期的“秒”部分，值为 0～59。
- getTime 函数：返回系统时间，具体为 1970 年 1 月 1 日至当前时间的毫秒数。

下面是一个使用以上日期函数的代码示例(详见源代码 ch09 目录中的 ch09-es-date.html 文件)。

【代码 9-11】

```
01 <script type="text/javascript">
02  var date = new Date();
03  console.log("getYear() is " + date.getYear());
04  var thisYear = 1900 + date.getYear();
05  console.log("This year is " + thisYear);
06  console.log("getMonth() is " + date.getMonth());
07  var thisMonth = date.getMonth() + 1;
08  console.log("This month is " + thisMonth);
09  console.log("getDate() is " + date.getDate());
10  var thisDate = date.getDate();
11  console.log("This date is " + thisDate);
12  console.log("getDay() is " + date.getDay());
13  console.log("getHours() is " + date.getHours());
14  console.log("getMinutes() is " + date.getMinutes());
15  console.log("getTime() is " + date.getTime());
16 </script>
```

运行 HTML 页面，控制台输出的调试信息如图 9.13 所示。

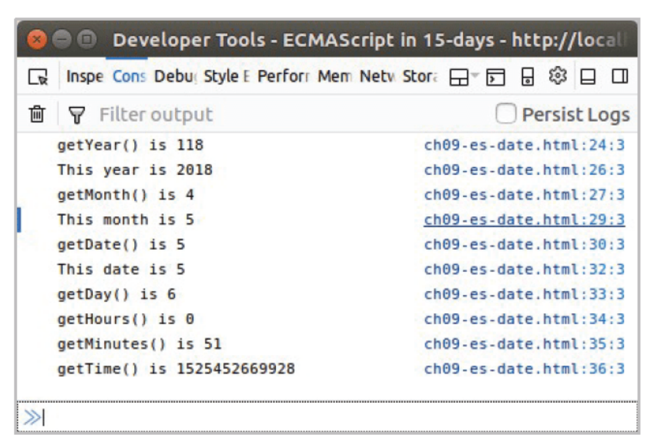


图 9.13　ECMAScript 日期函数

9.6　本章小结

本章主要介绍了 ECMAScript 语法规范中关于系统函数的知识，包括常规函数、字符串函数、数学函数、数组函数和日期函数等，并通过一些具体实例进行了讲解。

第 10 章

函数扩展

本章将介绍 ECMAScript 语法规范中关于函数扩展的内容，主要包括 ECMAScript 6 版本中新增的特性。ECMAScript 函数扩展主要包括函数的参数扩展、属性扩展和运算符扩展等知识点。

10.1　ECMAScript 函数参数扩展

ECMAScript 6 语法规范中为函数参数扩展了参数默认值、不定参数和 rest 参数等几个新功能，以解决 ECMAScript 早期版本中函数参数存在的问题。

10.1.1　可变参数

严格来说，可变参数是借助于 ECMAScript 函数的 arguments 对象来实现的。通过使用可变参数，可以模拟实现函数重载的功能，这也是 ECMAScript 脚本语言具有“面向对象”特性的标志之一。

下面是一个通过 ECMAScript 函数的可变参数查询匹配子字符串的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-varparams.html 文件）。

【代码 10-1】

```
01  <script type="text/javascript">
02      'use strict';
03      /*
04       * 定义 testSubStr 函数，检查是否包括全部子字符串
05       */
06      function testSubStr(oriStr) {
07          for (var i = 1; i < arguments.length; i++) {
08              var subStr = arguments[i];
09              if (oriStr.indexOf(subStr) === -1) {
10                  return false;
```

```
11                }
12            }
13            return true;
14        }
15        console.log("--- Test 'AS' in 'ECMAScript' ---");
16        if (testSubStr("ECMAScript", "AS"))
17            console.log("'ECMAScript' includes 'AS'.");
18        else
19            console.log("'ECMAScript' does not includes 'AS'");
20        console.log();
21        console.log("--- Test 'AS' & 'Sc' in 'ECMAScript' ---");
22        if (testSubStr("ECMAScript", "AS", "Sc"))
23            console.log("'ECMAScript' includes 'AS' and 'Sc'.");
24        else
25            console.log("'ECMAScript' does not includes 'AS' or 'Sc'");
26        console.log();
27        console.log("--- Test 'AS' & 'Sc' & 'Cr' in 'ECMAScript' ---");
28        if (testSubStr("ECMAScript", "AS", "Sc", "Cr"))
29            console.log("'ECMAScript' includes 'AS' & 'Sc' & 'Cr'.");
30        else
31            console.log("'ECMAScript' does not includes 'AS', 'Sc' or 'Cr'");
32        console.log();
33    </script>
```

关于【代码 10-1】的分析如下：

第 05～13 行代码定义了一个函数方法（testSubStr(oriStr)），用于检查某个原始字符串中是否包括给定的子字符串（数量是一个或多个）。该函数方法仅仅定义了一个参数（oriStr），用来传递原始字符串。那么如何传递给定的全部需要检查的子字符串呢?

第 06 和第 07 行代码通过 for 循环语句获取了 arguments 对象(数组)的全部数组项，该 arguments 对象包括全部给定的子字符串。

第 08 行代码通过 indexOf()函数方法判断给定的子字符串是否包含在原始字符串中，只要有一个不包括就返回 false。当全部包括在其中时，才会通过第 12 行代码返回 true。

第 15～19、21～25 和 27～31 行代码分别对 testSubStr(oriStr)函数方法进行了测试，第一个测试方法传递了两个参数，第二个测试方法传递了三个参数，第三个测试方法传递了四个参数。

运行页面，控制台输出的调试信息如图 10.1 所示。第一个和第二个测试方法成功检测到了子字符串，第三个测试方法提示子字符串 Cr 没有被检测到。同时，虽然 testSubStr(oriStr)函数方法仅仅定义了一个参数，但通过 arguments 对象是可以传入多个参数的。

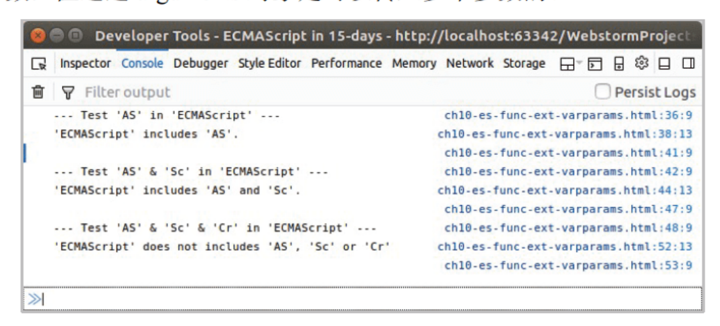


图 10.1 ECMAScript 可变参数

另外，虽然 testSubStr(oriStr)函数方法实现了可变参数特性，但其可读性明显要差很多。因此，ECMAScript 6 语法规范中新增了一个 rest 参数，用来替换可变参数所实现的功能。

10.1.2　rest 参数

ECMAScript 6 语法规范中为函数扩展了一个 rest 参数的功能，对于实现可变参数功能提供了更好的解决方案。使用 rest 参数方法时，该参数一定要放在参数列表的末尾，且必须使用连续 3 个小数点符号（...）开头。使用 rest 参数方法的代码可读性更好，同时解决了对 arguments 对象依赖的问题。

下面是一个根据【代码 10-1】改写而成、基于 rest 参数方法的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-rest.html 文件）。

【代码 10-2】

```
01  <script type="text/javascript">
02      'use strict';
03      /*
04       * 定义 testSubStr 函数，检查是否包括任意一个子字符串
05       */
06      function testSubStr(oriStr, ...args) {
07          let bFind = false;
08          for (let subStr of args) {
09              if (oriStr.indexOf(subStr) !== -1) {
10                  console.log("find '" + subStr + "' ...");
11                  bFind = true;
12              }
13          }
14          return bFind;
15      }
16      console.log("--- Test 'AS' in 'ECMAScript' ---");
17      if (testSubStr("ECMAScript", "AS"))
18          console.log("'ECMAScript' includes 'AS'.");
19      else
20          console.log("'ECMAScript' does not include 'AS'");
21      console.log("--- Test 'aS' or 'sC' in 'ECMAScript' ---");
22      if (testSubStr("ECMAScript", "aS", "sC"))
23          console.log("'ECMAScript' includes 'aS' or 'sC'.");
24      else
25          console.log("'ECMAScript' does not include 'aS' or 'sC'.");
26      console.log("--- Test 'AS' & 'Sc' & 'Cr' in 'ECMAScript' ---");
27      if (testSubStr("ECMAScript", "AS", "Sc", "Cr"))
28          console.log("'ECMAScript' includes 'AS', 'Sc' or 'Cr'.");
29      else
30          console.log("'ECMAScript' does not include 'AS', 'Sc' or 'Cr'");
31      console.log();
32  </script>
```

关于【代码 10-2】的分析如下：

第 06～15 行代码定义了一个 testSubStr(oriStr, ...args)函数。注意，该函数的第二个参数“...args”开始的连续 3 个小数点符号（.）表明该参数是一个 rest 参数。另外，在 testSubStr()函数内定义了

一个布尔值标志，作为函数的返回值。

【代码 10-2】与【代码 10-1】主要的区别就是，【代码 10-2】是检查源字符串中是否包括任意一个定义的子字符串，只要包括任意一个就返回 true，都不包括才会返回 false。

运行页面，控制台输出的调试信息如图 10.2 所示。从浏览器控制台中输出的结果来看，rest 参数与可变参数功能一样，很好地解决了 ECMAScript 函数中可变参数的问题。

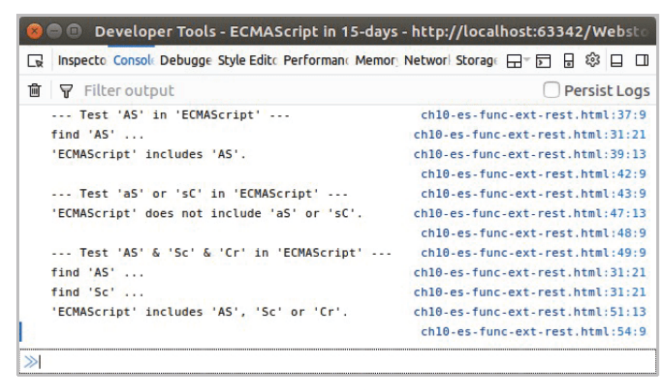


图 10.2　ECMAScript 中 rest 参数的应用

10.1.3　参数默认值

ECMAScript 6 语法规范中新增了关于“参数默认值”的概念，为没有传递参数值的函数方法提供了避免错误的解决方案。其实，在 ECMAScript 语法规范中还没有“参数默认值”这个方式时，设计人员均会采用编写代码的方式为参数传递默认值，这样就会避免出现不可预知的代码解析异常问题。

下面先看一个如何通过人工方式实现参数默认值的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-params-manual.html 文件）。

【代码 10-3】

```
01  <script type="text/javascript">
02      'use strict';
03      /*
04       * 定义 rectArea 函数，计算矩形的面积
05       */
06      function rectArea(length, width) {
07          length = length || 0;
08          width = width || 0;
09          return length * width;
10      }
11      console.log("--- rectArea() ---");
12      console.log("rectArea() = " + rectArea());
13      console.log("--- rectArea(30) ---");
14      console.log("rectArea(30) = " + rectArea(30));
15      console.log("--- rectArea(30, 60) ---");
16      console.log("rectArea(30, 60) = " + rectArea(30, 60));
17  </script>
```

关于【代码 10-3】的分析如下：

第 06～10 行代码定义了一个 rectArea(length, width)函数方法，用于计算矩形的面积。

第 07 和第 08 行代码通过“或”运算为参数 length 和 width 传递了默认值，这两行代码的处理方式就是人工定义参数默认值的方法。

第 09 行代码将计算的矩形面积值进行返回。

运行页面，控制台输出的调试信息如图 10.3 所示。即使调用 rectArea(length, width)函数方法时未定义默认参数值，也一样可以得出矩形面积的默认值。

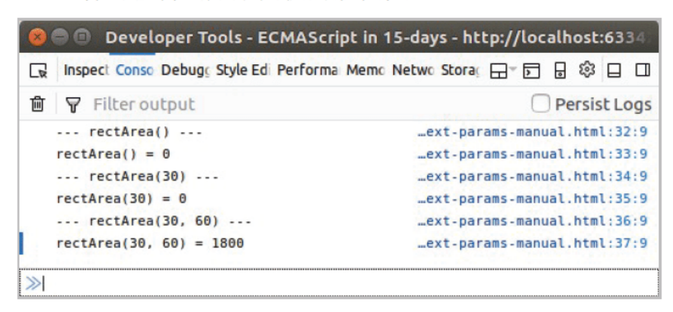


图 10.3　人工方式的参数默认值

下面再看一个在 ECMAScript 6 语法规范中使用参数默认值的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-params-default.html 文件）。

【代码 10-4】

```
01  <script type="text/javascript">
02      'use strict';
03      /*
04       * 定义 rectArea 函数，计算矩形的面积
05       */
06      function rectArea(length=100, width=100) {
07          return length * width;
08      }
09      console.log("--- rectArea() ---");
10      console.log("rectArea() = " + rectArea());
11      console.log("--- rectArea(50) ---");
12      console.log("rectArea(50) = " + rectArea(50));
13      console.log("--- rectArea(30, 60) ---");
14      console.log("rectArea(30, 60) = " + rectArea(30, 60));
15  </script>
```

关于【代码 10-4】的分析如下：

第 06～08 行代码定义了一个 rectArea(length, width)函数方法，用于计算矩形的面积。在 06 行函数方法的定义中，直接为参数定义了默认参数值 length=100、width=100，这就是 ECMAScript 6 语法规范中参数默认值的定义方法。

第 07 行代码返回计算的矩形面积值。

运行页面，控制台输出的调试信息如图 10.4 所示。从浏览器控制台中输出的结果来看，ECMAScript 6 语法规范中的参数默认值方式与人工默认参数方式（参考图 10.3）的功能是一致的，但参数默认值方式明显更为简洁。

```
--- rectArea() ---                 …ext-params-default.html:30:9
rectArea() = 10000                 …ext-params-default.html:31:9
--- rectArea(50) ---               …ext-params-default.html:32:9
rectArea(50) = 5000                …ext-params-default.html:33:9
--- rectArea(30, 60) ---           …ext-params-default.html:34:9
rectArea(30, 60) = 1800            …ext-params-default.html:35:9
```

图 10.4　ECMAScript 6 参数默认值方式

10.1.4　省略参数默认值的正确方式

ECMAScript 6 语法规范中新增的参数默认值固然非常好用，但使用方式不当也会带来错误问题。产生错误的原因主要是由于“非尾部的参数默认值”无法省略，而且在实际应用中也很难注意到这个问题。

下面看一个关于省略参数默认值的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-params-default-right.html 文件）。

【代码 10-5】

```
01   <script type="text/javascript">
02       'use strict';
03       /*
04        * 定义 rectArea 函数，计算矩形的面积
05        */
06       function rectArea(length=100, width=100) {
07           return length * width;
08       }
09       console.log("--- rectArea(60, ) ---");
10       console.log("rectArea(60, ) = " + rectArea(60, ));
11   </script>
```

关于【代码 10-5】的分析如下：

第 06～08 行代码定义了一个 rectArea(length=100, width=100)函数方法，用于计算矩形的面积，这个函数与【代码 10-4】中的定义是完全相同的。

第 09 和第 10 行代码直接调用 rectArea(60,)函数。注意，这里省略了第二个参数的默认值。

运行页面，控制台输出的调试信息如图 10.5 所示。省略了第二个参数的默认值且获取了默认值 100，同时也得出正确的计算结果。

如果尝试在【代码 10-5】中省略第一个参数的默认值，那么会得到同样的效果吗？

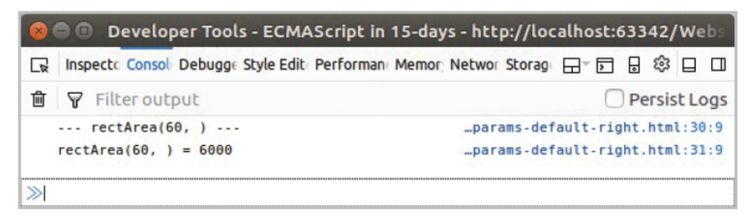


图 10.5　ECMAScript 6 省略参数默认值的正确方式（1）

下面继续看一个将【代码 10-5】稍作修改后的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-params-default-right.html 文件）。

【代码 10-6】

```
01  <script type="text/javascript">
02      'use strict';
03      /*
04       * 定义 rectArea 函数，计算矩形的面积
05       */
06      function rectArea(length=100, width=100) {
07          return length * width;
08      }
09      console.log("--- rectArea(, 60) ---");
10      console.log("rectArea(, 60) = " + rectArea(, 60));
11  </script>
```

关于【代码 10-6】的分析如下：

这段代码与【代码 10-5】唯一的不同就是省略了第一个参数的默认值，而没有省略第二个参数的默认值。

运行页面，控制台输出的调试信息如图 10.6 所示。在省略了第一个参数的默认值而没有省略第二个参数的默认值后，浏览器控制台调试器提示了错误。

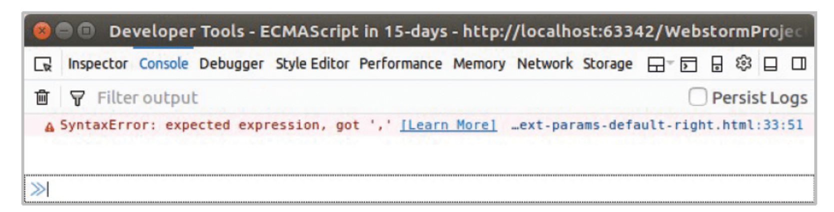


图 10.6　ECMAScript 6 省略参数默认值的正确方式（2）

如何避免出现省略“非尾部的参数默认值”产生的错误呢？

下面继续看一个将【代码 10-6】稍作修改后的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-params-default-right.html 文件）。

【代码 10-7】

```
01  <script type="text/javascript">
02      'use strict';
03      /*
04       * 定义 rectArea 函数，计算矩形的面积
05       */
06      function rectArea(length=100, width=100) {
07          return length * width;
08      }
09      console.log("--- rectArea(undefined, 60) ---");
10      console.log("rectArea(undefined, 60) = " + rectArea(undefined, 60));
11  </script>
```

关于【代码 10-7】的分析如下：

这段代码与【代码 10-6】唯一的不同就是，将省略的第一个参数默认值设置为了特殊值 undefined。

运行页面，控制台输出的调试信息如图 10.7 所示。在将省略的第一个参数默认值替换为特殊值 undefined 后，得到了正确的计算结果。

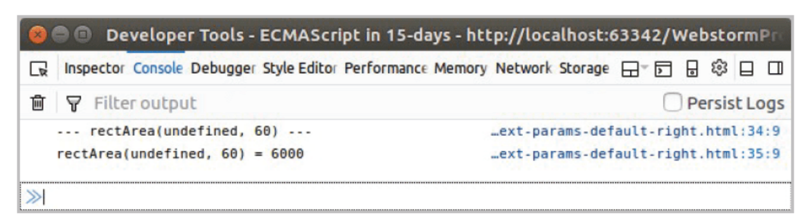


图 10.7　ECMAScript 6 省略参数默认值的正确方式（3）

10.2　length 属性扩展

ECMAScript 6 语法规范中为 ECMAScript 函数的 length 属性扩展了定义，用来增强 length 属性的用途。

10.2.1　参数默认值方式下的 length 属性

我们知道通过 ECMAScript 函数的 length 属性可以返回参数的数量。如果指定了参数的默认值，length 属性就会忽略这些已经指定了默认值的参数。

下面是一个在定义了参数默认值后获取 length 属性的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-length-default.html 文件）。

【代码 10-8】

```
01  <script type="text/javascript">
02       /*
03        * ECMAScript 函数 length 属性
04        */
05       console.log("func_length() length : ");
06       console.log((function func_length() {}).length);
07     console.log("func_length(a) length : ");
08     console.log((function func_length(a) {}).length);
09     console.log("func_length(a, b = 1) length : ");
10     console.log((function func_length(a, b = 1) {}).length);
11  </script>
```

关于【代码 10-8】的分析如下：

第 05 和第 06 行代码测试了函数无参数时的 length 属性值。

第 07 和第 08 行代码测试了函数带参数时的 length 属性值。

第 09 和第 10 行代码测试了函数参数带默认值时的 length 属性值。

运行页面，控制台输出的调试信息如图 10.8 所示。若函数参数定义了默认值，则在使用 length 属性时是不计入函数参数数量的。

```
func_length() length :
0
func_length(a) length :
1
func_length(a, b = 1) length :
1
```

图 10.8　ECMAScript length 属性扩展（1）

10.2.2　rest 参数方式下的 length 属性

若 ECMAScript 函数使用了 rest 参数的方式，则 length 属性同样会忽略已经指定了默认值的参数，这主要因为 length 属性表示该函数预期传入的参数个数。

下面是一个在 rest 参数默认值方式下获取 length 属性的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-length-rest.html 文件）。

【代码 10-9】

```
01  <script type="text/javascript">
02      /*
03       * ECMAScript 函数 length 属性
04       */
05      console.log("func_length(...args) length : ");
06      console.log((function func_length(...args) {}).length);
07      console.log("func_length(a, ...args) length : ");
08      console.log((function func_length(a, ...args) {}).length);
09      console.log("func_length(a, b, ...args) length : ");
10      console.log((function func_length(a, b, ...args) {}).length);
11  </script>
```

关于【代码 10-9】的分析如下：

这段代码主要测试了在使用 rest 参数的方式下 length 属性的取值情况。

运行页面，控制台输出的调试信息如图 10.9 所示。若函数参数使用了 rest 参数方式，则在使用 length 属性时 rest 参数同样是不计入函数参数数量的。

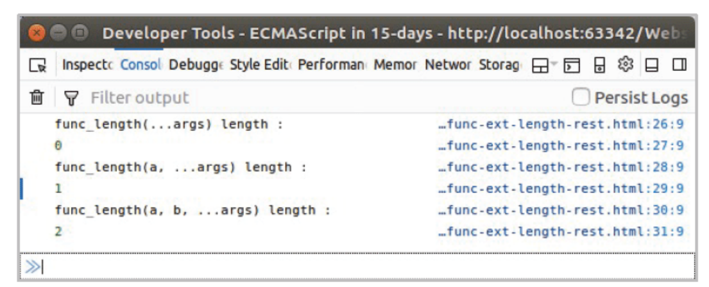


图 10.9　ECMAScript length 属性扩展（2）

10.2.3 参数默认值不同位置下的 length 属性

前面介绍了如果在指定了参数的默认值后，length 属性就会忽略这些已经指定了默认值的参数。同时，如果参数默认值定义在不同的位置，就会影响到 length 属性对后面参数的计数。

下面是一个参数默认值在不同位置下使用 length 属性的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-length-params.html 文件）。

【代码 10-10】

```
01  <script type="text/javascript">
02      /*
03       * ECMAScript 函数 length 属性
04       */
05      console.log("func_length_num(a=1, b, c) length : ");
06      console.log((function func_length_first(a=1, b, c) {}).length);
07      console.log("func_length_second(a, b=2, c) length : ");
08      console.log((function func_length_second(a, b=2, c) {}).length);
09      console.log("func_length_third(a, b, c=3) length : ");
10      console.log((function func_length_third(a, b, c=3) {}).length);
11  </script>
```

关于【代码 10-10】的分析如下：

这段代码主要测试了参数默认值在不同位置时 length 属性的取值情况。

运行页面，控制台输出的调试信息如图 10.10 所示。若函数的参数默认值没有定义在参数序列的结尾，则 length 属性会将后面的参数不做计数。

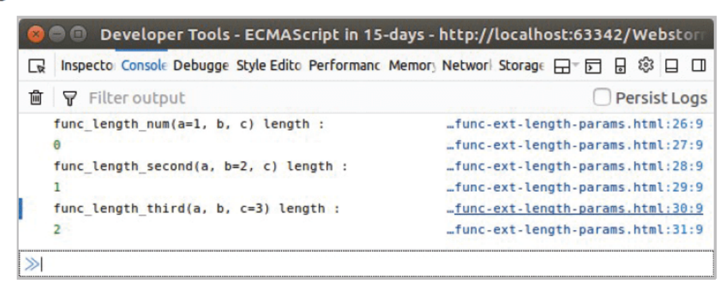


图 10.10 ECMAScript length 属性扩展（3）

10.3 name 属性扩展

ECMAScript 6 语法规范中为 ECMAScript 函数的 name 属性扩展了定义，虽然从 ECMAScript 早期版本开始已经开始支持 name 属性了，但直到 ECMAScript 6 版本开始才将 name 属性写入正式标准。

一般来说，通过 name 属性可以获取 ECMAScript 函数的名称。另外，在使用匿名函数的情形下，早期版本只会返回一个空字符串。

下面是一个使用 name 属性获取函数名称的代码示例（详见源代码 ch10 目录中的

ch10-es-func-ext-name.html 文件）。

【代码 10-11】

```
01   <script type="text/javascript">
02        'use strict';
03       /*
04        * ECMAScript 函数扩展 --- name 属性
05        */
06       function func_name() {}
07       console.log("--- function func_name() {} ---");
08       console.log(func_name.name);
09       var f_name = function () {};
10       console.log("--- f_name = function () {} ---");
11       console.log(f_name.name);
12   </script>
```

关于【代码 10-11】的分析如下：

第 06 行代码定义了第一个函数 func_name()，并在第 08 行代码中通过函数名调用了 name 属性。

第 09 行代码定义了第二个函数（匿名函数），并赋值给了变量 f_name；第 11 行代码中通过变量 f_name 调用了 name 属性。

运行页面，控制台输出的调试信息如图 10.11 所示。通过函数名称（匿名函数通过变量名）调用 name 属性，可以获取该函数的名称。

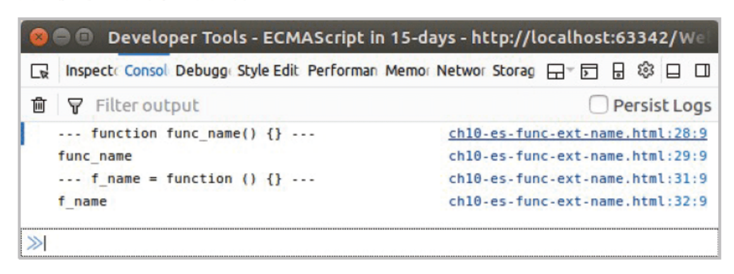


图 10.11　ECMAScript name 属性扩展

10.4　箭头函数

ECMAScript 6 语法规范中新增加了一个比较有意思的箭头函数功能，用来扩展对 ECMAScript 函数的使用方式。

10.4.1　箭头函数的基本形式

ECMAScript 6 版本设计的箭头函数的样式看起来有点怪，但丝毫不影响其功能的强大。箭头函数的基本语法形式如下：

```
var 函数名 = (参数) => {函数体}     // 注意 "=>" 符号的使用
```

该语法形式大致描述了箭头函数的样式，总体还包括了函数定义的一些基本元素（如函数名、

参数和函数体）。另外，箭头函数在具体使用时很灵活，可以表现为多种样式。

下面是一个使用 ECMAScript 箭头函数基本形式的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-arrow-basic.html 文件）。

【代码 10-12】

```
01  <script type="text/javascript">
02       'use strict';
03      /*
04       * ECMAScript 函数
05       */
06      function func_basic(p) {
07           return p;
08      }
09      var v = func_basic("ECMAScript");
10      console.log("func_basic : " + v);
11      /*
12       * ECMAScript 函数扩展 --- 箭头函数
13       */
14      var func_arrow_basic = p => p;
15      var a = func_arrow_basic("ECMAScript");
16      console.log("func_arrow_basic : " + a);
17  </script>
```

关于【代码 10-12】的分析如下：

第 06～08 行代码定义了一个普通函数 func_basic()，第 09～10 行代码是对该函数的调用。

第 14 行代码定义了一个箭头函数：函数名为 func_arrow_basic，参数为 p，函数体为返回参数 p。

第 15 行代码是对箭头函数的调用，与第 09 行代码定义的普通函数的调用方式是一致的。

运行页面，控制台输出的调试信息如图 10.12 所示。从浏览器控制台输出的结果来看，第 14 行代码定义的箭头函数 func_arrow_basic()，在功能上完全等同于第 06～08 行代码定义的一个普通函数 func_basic()。

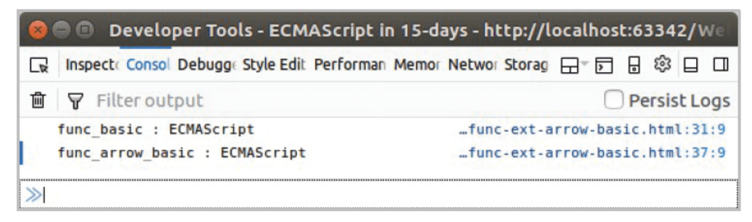


图 10.12　ECMAScript 箭头函数（1）

10.4.2　箭头函数的参数

如果定义的箭头函数不带参数或多于一个参数时，就要使用小括号“()”将参数包括进去。只有在仅仅定义了一个参数的情况下才可以省略小括号，如【代码 10-12】中箭头函数的定义。

下面是一个使用 ECMAScript 箭头函数时不带参数的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-arrow-noparam.html 文件）。

【代码 10-13】

```
01  <script type="text/javascript">
02       'use strict';
03      /*
04       * ECMAScript 函数扩展 --- 箭头函数
05       */
06      var func_arrow_noparam = () => "ECMAScript";
07      var n = func_arrow_noparam();
08      console.log("func_arrow_noparam : " + n);
09  </script>
```

关于【代码 10-13】的分析如下：

第 06 行代码定义了一个箭头函数：函数名为 func_arrow_noparam，参数为一个空括号（表示无参数），函数体为直接返回一个字符串。

运行页面，控制台输出的调试信息如图 10.13 所示。

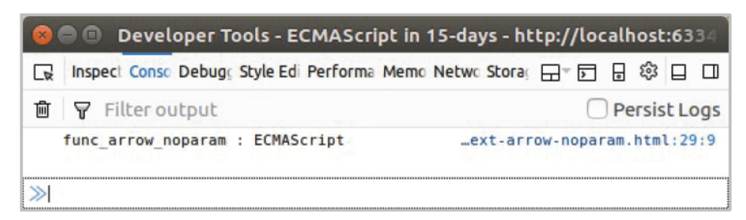


图 10.13　ECMAScript 箭头函数（2）

下面是一个使用 ECMAScript 箭头函数时带多个参数的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-arrow-params.html 文件）。

【代码 10-14】

```
01  <script type="text/javascript">
02       'use strict';
03      /*
04       * ECMAScript 函数扩展 --- 箭头函数
05       */
06      var func_arrow_params = (a, b) => a + b;
07      var s = func_arrow_params("Hello", "ECMAScript");
08      console.log("func_arrow_params : " + s);
09      var n = func_arrow_params(1, 2);
10      console.log("func_arrow_params : " + n);
11  </script>
```

关于【代码 10-14】的分析如下：

第 06 行代码定义了一个箭头函数：函数名为 func_arrow_params，参数为(a, b)，函数体为返回两个参数的和运算（a + b）。

第 07 行代码是对箭头函数的一个调用，参数为两个字符串。

第 09 行代码是对箭头函数的另一个调用，参数为两个数值。

运行页面，控制台输出的调试信息如图 10.14 所示。从浏览器控制台输出的结果来看，箭头函数的参数对于字符串和数字都是可以满足的。

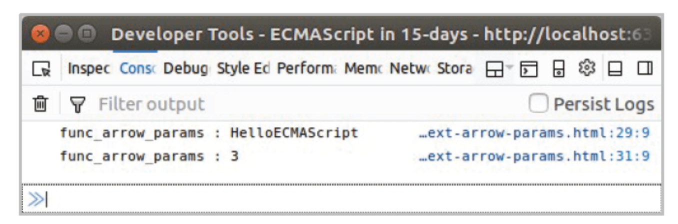


图 10.14　ECMAScript 箭头函数（3）

10.4.3　箭头函数的函数体

如果定义的箭头函数的函数体含有多条语句，就要使用大括号“{}”将语句块包括进去。

下面是一个使用 ECMAScript 箭头函数时不带参数的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-arrow-body.html 文件）。

【代码 10-15】

```
01  <script type="text/javascript">
02       'use strict';
03      /*
04       * ECMAScript 函数扩展 --- 箭头函数
05       */
06      var func_arrow_body = (a, b) => {var s = a + b; return s;}
07      var s = func_arrow_body("Hello", "ECMAScript");
08      console.log("func_arrow_body : " + s);
09      var n = func_arrow_body(1, 2);
10      console.log("func_arrow_body : " + n);
11  </script>
```

关于【代码 10-15】的分析如下：

第 06 行代码定义了一个箭头函数：函数名为 func_arrow_body，参数为(a, b)，函数体使用大括号“{}”定义了一个语句块（包含变量定义及“+”运算符的运算、计算结果的返回等语句）。

运行页面，控制台输出的调试信息如图 10.15 所示。

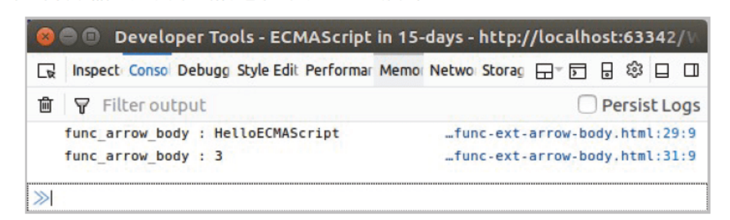


图 10.15　ECMAScript 箭头函数（4）

10.5　箭头函数扩展应用

ECMAScript 6 语法规范中新增的箭头函数具有很多的使用方式，可以扩展出很简洁且强大的功能。

10.5.1　箭头函数计算工具

通过编写一行箭头函数的代码，就可以实现传统方式下编写好几行代码才能实现的功能。例如，可以通过箭头函数编写出很实用的简单计算工具。

下面是一个使用 ECMAScript 箭头函数编写简单计算工具的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-arrow-cal.html 文件）。

【代码 10-16】

```
01  <script type="text/javascript">
02       'use strict';
03      /*
04       * ECMAScript 箭头函数 --- 计算矩形面积工具
05       */
06      var func_arrow_rect = (l, w) => l * w;
07      var areaRect = func_arrow_rect(6, 3);
08      console.log("矩形面积工具: func_arrow_rect(6, 3) = " + areaRect);
09      /*
10       * ECMAScript 箭头函数 --- 计算圆面积工具
11       */
12      var func_arrow_circle = r => (Math.PI * r * r).toFixed(2);
13      var areaCircle = func_arrow_circle(5);
14      console.log("圆形面积工具: func_arrow_circle(5) = " + areaCircle);
15      /*
16       * ECMAScript 箭头函数 --- 计算余数工具
17       */
18      var func_arrow_remainder = (i, j) => i % j;
19      var rd = func_arrow_remainder(8, 5);
20      console.log("计算余数工具: func_arrow_remainder(8, 5) = " + rd);
21  </script>
```

关于【代码 10-16】的分析如下：

第 06 行代码定义了一个计算矩形面积的箭头函数 func_arrow_rect()，第 07 和第 08 行代码是对该函数的调用。

第 12 行代码定义了一个计算圆形面积的箭头函数 func_arrow_circle()，第 13 和第 14 行代码是对该函数的调用。

第 18 行代码定义了一个计算余数的箭头函数 func_arrow_remainder()，第 19 和第 20 行代码是对该函数的调用。

运行页面，控制台输出的调试信息如图 10.16 所示。通过使用一行箭头函数代码就可以实现传统函数方式好几行代码才能实现的功能，箭头函数的简洁性和高效性十分明显。

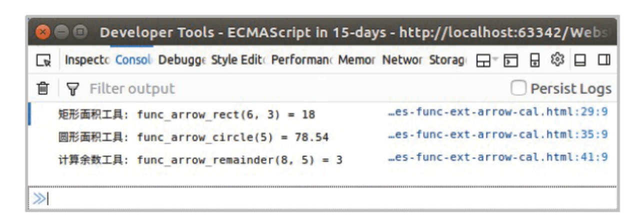


图 10.16　ECMAScript 箭头函数计算工具

10.5.2 箭头函数与解构赋值

通过将箭头函数与前文中介绍的解构赋值相结合，可以简化对箭头函数调用的方式，提高代码编写的效率。

下面是一个结合使用 ECMAScript 箭头函数与解构赋值的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-arrow-destructuring.html 文件）。

【代码 10-17】

```
01  <script type="text/javascript">
02       'use strict';
03      /*
04       * ECMAScript 箭头函数 --- 计算余数工具
05       */
06      var func_arrow_remainder = ([i, j]) => i % j;
07      var rd = func_arrow_remainder([8, 5]);
08      console.log("计算余数工具: func_arrow_remainder([8, 5]) = " + rd);
09  </script>
```

关于【代码 10-17】的分析如下：

这段代码主要就是将【代码 10-16】中的余数计算工具通过数组解构赋值的方式重写了一遍。

运行页面，控制台输出的调试信息如图 10.17 所示。

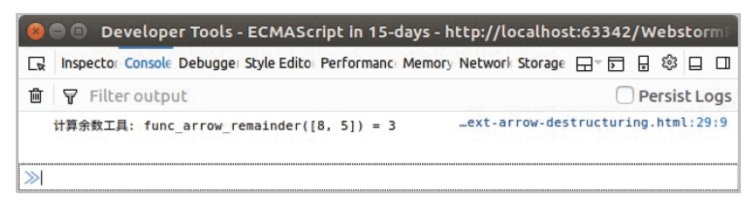


图 10.17　ECMAScript 箭头函数与解构赋值（1）

下面是另一个结合使用 ECMAScript 箭头函数与解构赋值的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-arrow-destructuring-rest.html 文件）。

【代码 10-18】

```
01  <script type="text/javascript">
02       'use strict';
03      /*
04       * ECMAScript 箭头函数 --- 最大值和最小值工具
05       */
06      var func_arrow_max = (...args) => Math.max(...args);
07      var max = func_arrow_max(...[1, 2, 3]);
08      console.log("最大值工具: func_arrow_max(...[1, 2, 3]) = " + max);
09      var func_arrow_min = (...args) => Math.min(...args);
10      var min = func_arrow_min(...[3, 2, 1]);
11      console.log("最小值工具: func_arrow_min(...[3, 2, 1]) = " + min);
12  </script>
```

关于【代码 10-18】的分析如下：

这段代码是通过箭头函数与数组解构赋值实现了最大值和最小值的计算工具，特殊之处就是使用到了前文中介绍过的 rest 参数方式来传递数组。

运行页面，控制台输出的调试信息如图 10.18 所示。

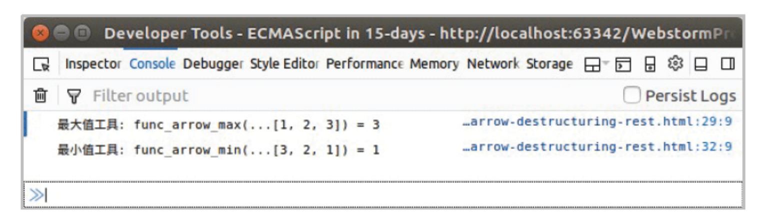


图 10.18　ECMAScript 箭头函数与解构赋值（2）

10.5.3　箭头函数与回调函数

通过将箭头函数与回调函数相结合，可以简化对回调函数的使用方式，提高代码编写的效率。

下面是一个通过 ECMAScript 箭头函数简化回调函数计算平方和的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-arrow-callback-sqr.html 文件）。

【代码 10-19】

```
01  <script type="text/javascript">
02      'use strict';
03      /*
04       * ECMAScript 箭头函数 --- 回调函数
05       */
06      var s = [1,3,5].map(function (i) {
07          return i * i;
08      });
09      console.log("Square of [1,3,5] : " + s);
10      var arrowS = [1,3,5].map(i => i * i);
11      console.log("Square of [1,3,5] by Arrow Func : " + arrowS);
12  </script>
```

关于【代码 10-19】的分析如下：

这段代码通过结合使用 map()函数方法和回调函数计算数值（1、3 和 5）平方和，主要是看一下通过箭头函数方式来简写回调函数的过程。

第 06～08 行代码通过 map()函数方法和回调函数实现了计算平方和的操作，这里的回调函数使用的是传统方式。

第 10 行代码同样是通过 map()函数方法和回调函数实现了计算平方和的操作，而这里的回调函数使用的是箭头函数方式。

运行页面，控制台输出的调试信息如图 10.19 所示。

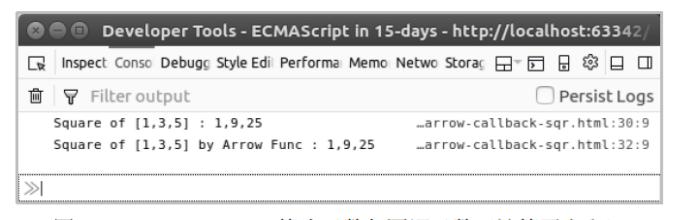


图 10.19　ECMAScript 箭头函数与回调函数（计算平方和）

下面是一个通过 ECMAScript 箭头函数简化回调函数进行数组排序的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-arrow-callback-sort.html 文件）。

【代码 10-20】

```
01  <script type="text/javascript">
02       'use strict';
03      /*
04       * ECMAScript 箭头函数 --- 回调函数
05       */
06      var arr = [7,3,9,5,1].sort(function (a, b) {
07           return a - b;
08      });
09      console.log("sort [7,3,9,5,1] : " + arr);
10      var arrowArr = [7,3,9,5,1].sort((a, b) => a - b);
11      console.log("sort [7,3,9,5,1] by Arrow Func : " + arrowArr);
12  </script>
```

关于【代码 10-20】的分析如下：

这段代码主要是通过结合使用 sort()函数方法和回调函数对无序数组[7,3,9,5,1]进行排序，同样也是看一下通过箭头函数方式来简写回调函数的过程。

第 06～08 行代码通过 sort()函数方法和回调函数实现了数组排序的操作，这里的回调函数使用的是传统方式。

第 10 行代码同样是通过 sort()函数方法和回调函数实现了数组排序的操作，而这里的回调函数使用的就是箭头函数方式。

运行页面，控制台输出的调试信息如图 10.20 所示。

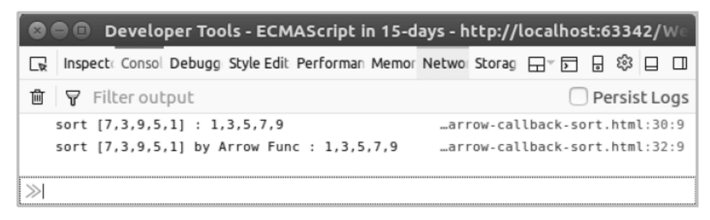


图 10.20　ECMAScript 箭头函数与回调函数（数组排序）

10.5.4　箭头函数与链式函数

将箭头函数与链式函数相结合，可以通过一行代码实现很复杂的功能，极大地提高了代码编写的效率。

下面是一个通过结合 ECMAScript 箭头函数与链式函数实现数组排序并计算平方和的代码示例（详见源代码 ch10 目录中的 ch10-es-func-ext-arrow-callback-sort-sqr.html 文件）。

【代码 10-21】

```
01  <script type="text/javascript">
02       'use strict';
03      /*
04       * ECMAScript 箭头函数 --- 链式函数
05       */
```

```
06        console.log("传统方式 - 链式函数");
07        var arr = [7,3,9,5,1].map(function (i) {
08           return i * i;
09        }).sort(function (a, b) {
10           return a - b;
11         });
12        console.log("map + sort [7,3,9,5,1] : " + arr);
13        console.log("箭头函数方式 - 链式函数");
14        var arrowArr = [7,3,9,5,1].map(i => i * i).sort((a, b) => a - b);
15        console.log("map + sort [7,3,9,5,1] by Arrow Func : " + arrowArr);
16    </script>
```

关于【代码 10-21】的分析如下：

这段代码是通过链式函数方式调用 map()和 sort()函数方法对数组[7,3,9,5,1]进行排序与计算平方和，主要是看一下通过箭头函数方式来简写链式函数的过程。

第 07～11 行代码通过传统的链式函数调用方式，对数组进行了排序（sort()函数方法）与计算平方和（map()函数方法）的操作。

第 14 行代码通过箭头函数方式实现链式函数的调用，对数组进行了排序（sort()函数方法）与计算平方和（map()函数方法）的操作。

运行页面，控制台输出的调试信息如图 10.21 所示。

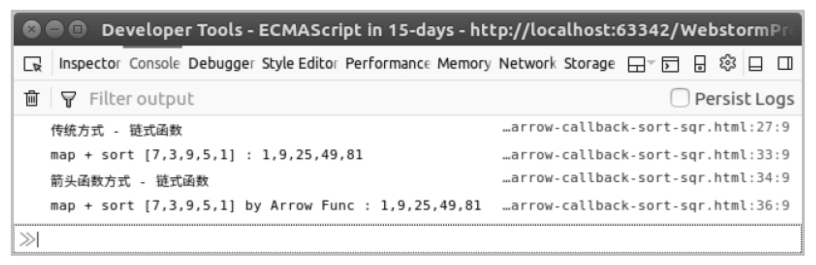


图 10.21　ECMAScript 箭头函数与链式函数（数组排序与计算平方和）

10.6　本章小结

本章主要介绍了 ECMAScript 语法规范中关于函数扩展方面的知识，包括 ECMAScript 函数的参数扩展、属性扩展、箭头函数等内容。本章整体属于 ECMAScript 6 版本语法规范中的新增内容，且理解有一定的难度，相信通过文中给出的代码实例可以帮助读者加深理解。

第 11 章

ECMAScript 对象

本章将介绍 ECMAScript 语法规范中关于对象（object）的基础内容，具体包括对象的创建、初始化、绑定及销毁等。对象（object）是 ECMAScript 语法规范中较为高级的内容，也是 ECMAScript 脚本语言中最核心的组成之一。

11.1　ECMAScript 对象

本节将介绍 ECMAScript 对象的基本知识，包括对象的概念、对象的构成及对象实例等，这些内容是学习 ECMAScript 对象编程的基础。

11.1.1　什么是 ECMAScript 对象

ECMA-262 规范将对象（object）定义为“属性的无序集合，每个属性存放一个原始值、对象或函数”。在 ECMAScript 脚本语言编程中有一种说法，就是“一切皆为对象”。这句话是什么含义呢？简单来说，就是 ECMAScript 脚本语言中的各种数据类型（如字符串、数字和数组）都是对象。

11.1.2　ECMAScript 对象构成

在 ECMAScript 语法规范中，对象（object）由特性（attribute）构成，特性可以是原始值、引用值或函数。如果特性定义为函数，那这个函数将作为对象的方法（method）来使用，否则该特性就作为对象的属性（property）来使用。

11.1.3　ECMAScript 对象实例

在使用 ECMAScript 对象创建变量并进行初始化后，这个变量就叫作对象的实例（instance），而创建实例的过程就叫作实例化（instantiation）。

11.2　创建 ECMAScript 对象

因为 ECMAScript 语法规范中对象原型定义为 Object 类型，所以可以通过 Object 类型创建 ECMAScript 对象。在具体声明一个 ECMAScript 对象时，是通过 new 关键字来创建的。

下面是一个通过 new 关键字创建 ECMAScript 对象的代码示例（详见源代码 ch11 目录中的 ch11-es-obj-new.html 文件）。

【代码 11-1】

```
01  <script type="text/javascript">
02      var obj = new Object();
03      var objStr = new String();
04  </script>
```

关于【代码 11-1】的分析如下：

第 02 行代码通过 new 关键字创建了一个 Object 类型实例，并存储到变量 obj 中。

第 03 行代码通过 new 关键字创建了一个 String 类型实例，并存储到变量 objStr 中。

若构造函数无参数，则括号不是必须使用的，这样【代码 11-1】可以采用下面的形式进行重写（详见源代码 ch11 目录中的 ch11-es-obj-new-omit.html 文件）。

【代码 11-2】

```
01  <script type="text/javascript">
02      var obj = new Object;
03      var objStr = new String;
04  </script>
```

【代码 11-2】与【代码 11-1】虽然写法略有区别，但在功能上是完全一致的。

11.3　ECMAScript 对象初始化

ECMAScript 对象创建完成后，其实只是在内存预留了空间（还未实际分配），真正的工作要放在对象初始化中。ECMAScript 对象的初始化是通过大括号“{}”来定义的，在大括号“{}”内部，对象的属性是通过“名称”和“值”对的形式（name : value）来定义的，其中“名称”和“值”用冒号“:”分隔。

下面是一个创建并初始化 ECMAScript 对象的代码示例（详见源代码 ch11 目录中的

ch11-es-obj-new-inst.html 文件）。

【代码 11-3】

```
01  <script type="text/javascript">
02      var userinfo = new Object();
03      userinfo = {
04          id : 123,
05          username : "tina",
06          email : "tina@email.com"
07      };
08      console.log(userinfo);
09  </script>
```

关于【代码 11-3】的分析如下：

第 02 行代码通过 new 关键字创建了 Object 类的一个实例，并将其存储到变量 userinfo，即 ECMAScript 对象中。

第 03～07 行代码对 userinfo 对象进行实例化，具体定义了 3 个属性（id、username 和 email），并进行了初始化赋值。

其实，第 03～07 行代码还可以写成一行代码的形式，具体如下：

```
var userinfo = { id : 123, username : "tina", email : "tina@email.com" };
```

这种写成一行的方式在代码较少的情况下是可行的，如果代码量很大，这样写就不适用了。

运行页面，控制台输出的调试信息如图 11.1 所示。

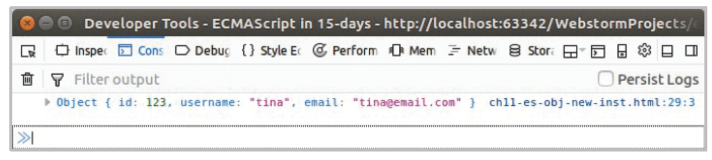


图 11.1　创建 ECMAScript 对象并初始化（1）

下面尝试将【代码 11-3】改写一下（详见源代码 ch11 目录中的 ch11-es-obj-new-inst-re.html 文件）。

【代码 11-4】

```
01  <script type="text/javascript">
02      var userinfo = new Object;
03      userinfo.id = 123;
04      userinfo.username = "tina";
05      userinfo.email = "tina@email.com";
06      console.log(userinfo);
07      var len = Object.keys(userinfo);
08      console.log("userinfo's length : " + len.length);
09  </script>
```

关于【代码 11-4】的分析如下：

第 02 行代码通过 new 关键字创建了 Object 类的一个实例，并将其存储到变量 userinfo，即 ECMAScript 对象中。注意，这里在声明对象时，Object 类没有加上括号。

第 03～05 行代码对 userinfo 对象进行实例化，具体定义了 3 个属性（id、username 和 email），

并进行了初始化赋值。

第 06 行代码通过 console.log()函数以调试信息的方式，在控制台中输出了第 03～05 行代码定义的 ECMAScript 对象 userinfo。

第 07 行代码中通过 Object 类型的 keys()方法获取了 ECMAScript 对象 userinfo 的键值对数量，并保存在变量 len 中。

第 08 行代码通过 len 变量的 length 属性，在控制台中输出了 ECMAScript 对象 userinfo 的长度。

运行页面，控制台输出的调试信息如图 11.2 所示。

图 11.2　创建 ECMAScript 对象并初始化（2）

11.4　ECMAScript 对象销毁

ECMAScript 脚本语言在设计上借鉴了很多先进的设计理念，比较著名的就是“无用存储单元收集程序（garbage collection routine）”，该理念主要借鉴了 Java 语言的“垃圾回收机制”。设计无用存储单元收集程序，就意味着程序员开发时不必专门考虑通过销毁对象来释放内存了。

无用存储单元收集程序的原理是，当代码中已经没有对某个对象的引用时，该对象将会被销毁。例如，每当一个函数执行完代码后，无用存储单元收集程序都会运行，然后将无效的对象进行销毁，从而释放无用的内存空间。当然，在某些特殊的、不可预知的情况下，无用存储单元收集程序也会运行。

ECMAScript 语法规范中也支持强制销毁对象，通过把对象的所有引用都定义为 null 原始值，就可以强制性地销毁对象。

下面是一个将 ECMAScript 对象进行销毁的代码示例（详见源代码 ch11 目录中的 ch11-es-obj-null.html 文件）。

【代码 11-5】

```
01  <script type="text/javascript">
02      var userinfo = {
03          id : 123,
04          username : "tina",
05          email : "tina@email.com"
06      };
07      console.log(userinfo);
08      userinfo = null;
09      console.log(userinfo);
10  </script>
```

关于【代 11-5】的分析如下：

第 02～06 行代码定义了一个对象 userinfo，并进行了实例化操作。

第 07 行代码先将对象 userinfo 的内容输出到浏览器控制台中进行显示。

第 08 行代码强制性将对象 userinfo 重新定义为 null。

第 09 行代码再次将强制销毁后的对象 userinfo 的内容输出到浏览器控制台中，这样可以与第 07 行代码输出的内容进行对比。

运行页面，控制台输出的调试信息如图 11.3 所示。如图 11.3 中箭头所指，在第 08 行代码强制销毁对象 userinfo 后，第 09 行代码再次输出该对象的内容为 null，说明此时对对象 userinfo 的引用已经不存在了。那么当无用存储单元收集程序再次运行时，该对象将被销毁。

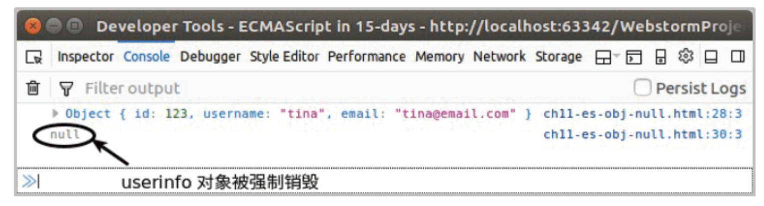


图 11.3　ECMAScript 对象被强制销毁（1）

这里还需要介绍一个概念，就是 ECMAScript 对象的引用。在 ECMAScript 语法规范中规定，代码不能访问对象的物理地址，能访问的仅仅是对象的引用。我们在使用 ECMAScript 语言创建对象时，都会定义一个变量用于存储该对象，这个变量就是对该对象的引用（但不是对象本身）。

下面继续将【代码 11-5】重新改写一下，实现一个将 ECMAScript 对象引用和 ECMAScript 对象销毁相结合的代码示例（详见源代码 ch11 目录中的 ch11-es-obj-null-re.html 文件）。

【代码 11-6】

```
01  <script type="text/javascript">
02      var userinfo = {
03         id : 123,
04         username : "tina",
05         email : "tina@email.com"
06      };
07      console.log("usreinfo inst :");
08      console.log(userinfo);
09      var userinfo_a = userinfo;  // TODO: 定义对象引用
10      console.log("userinfo_a = usreinfo :");
11      console.log(userinfo_a);
12      userinfo = null;
13      console.log("usreinfo = null :");
14      console.log(userinfo);
15      console.log("after usreinfo = null, userinfo_a :");
16      console.log(userinfo_a);
17  </script>
```

关于【代码 11-6】的分析如下：

第 02～08 行代码大体上沿用了【代码 6-5】中的代码。

第 09 行代码定义了第二个对象 userinfo_a，并通过赋值为 userinfo 再次引用了该对象。

第 11 行代码将对象 userinfo_a 的内容输出到浏览器控制台中进行显示。

第 12 行代码强制性将对象 userinfo 重新定义为 null。

第 14 行和第 16 行代码再次将对象 userinfo 和 userinfo_a 的内容输出到浏览器控制台中进行显示，目的是测试一下将对象 userinfo 强制性定义为 null 后，对象 userinfo_a 有没有受到影响。

运行页面，控制台输出的调试信息如图 11.4 所示。

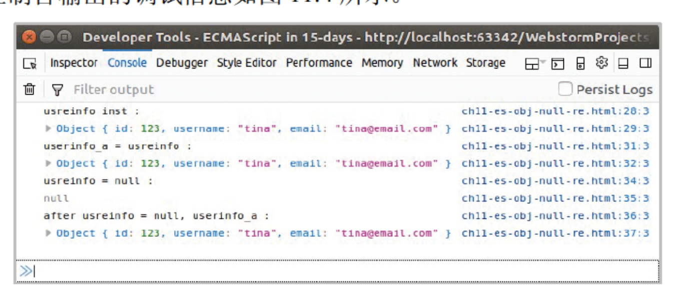


图 11.4　ECMAScript 对象强制性销毁（2）

如图 11.4 所示，从第 11 行代码输出的结果来看，第 09 行代码定义的第二个对象 userinfo_a 完全复制了第 02～06 行代码定义的第一个对象 userinfo。

在第 12 行代码强制性销毁对象 userinfo 后，从第 16 行代码输出的结果来看，第二个对象 userinfo_a 没有受到任何影响。

【代码 11-6】的测试结果充分验证了前文中对 ECMAScript 对象引用的描述，即在定义了对象的多个引用后，销毁其中一个并不会对其他对象引用产生影响。

11.5　ECMAScript 对象绑定方式

这里再介绍一下 ECMAScript 对象绑定的内容。根据 ECMA-262 官方文档的描述，绑定（binding）就是将对象的接口与对象实例结合在一起的过程。绑定（binding）一般分为早绑定（early binding）和晚绑定（late binding）两种方式。

早绑定（early binding）就是指在实例化对象之前定义它的属性和方法，这样编译器或解释程序就能提前将其转换机器代码。因为 ECMAScript 脚本语言不是强类型语言，所以不支持早绑定，而诸如 C 语言、C#语言和 Java 语言这类的，就支持早绑定。

晚绑定（late binding）就是指编译器或解释程序在运行前不知道对象的类型。因此使用晚绑定的语言无须检查对象的类型，只检查对象是否支持属性和方法即可。ECMAScript 脚本语言中的所有变量都采用晚绑定方式，这样有利于执行大量的对象操作，并且没有任何的限制。

11.6　本章小结

本章主要介绍了 ECMAScript 语法规范中关于对象的基础知识，包括对象的创建、初始化、销毁及绑定等方面的内容，并通过一些具体实例进行了讲解。

第 12 章

对象类型

在 ECMAScript 语法规范中，对象分为本地对象、内置对象和宿主对象三大类。本章将介绍本地对象和内置对象，本地对象属于 ECMA-262 规范定义的引用类型，而且内置对象严格来说也属于本地对象。

12.1 ECMAScript 对象概述

对于 ECMAScript 脚本语言来说，一切事物皆为对象。那么如何理解这句话呢？简单来说，Object 类型是对象，数值类型（Number）、字符串类型（String）、数组类型（Array）也是对象，而日期类型（Date）、正则表达式（RegExp）、甚至函数类型（Function）同样是对象，所以是“一切皆为对象”。

本地对象主要包括数值类型（Number）、字符串类型（String）、数组类型（Array）、日期类型（Date）、正则表达式（RegExp）等。ECMA-262 规范中对于本地对象有一个定义：“独立于宿主环境的、由 ECMAScript 语言实现所提供的对象”。因此，将本地对象理解为 ECMA-262 规范中所描述的类（引用类型）的概念更为恰当。

ECMAScript 语言还定义了两个内置对象，即 Global 和 Math。ECMA-262 规范中对于内置对象也有一个定义：“由 ECMAScript 语言实现所提供的、独立于宿主环境的所有对象，在 ECMAScript 程序开始执行时出现”。内置对象意味着设计人员不必明确实例化对象，其已被实例化了。

另外，ECMA-262 规范中定义了所有非本地对象都是宿主对象，宿主对象即是由 ECMAScript 的宿主环境所提供的对象，具体是指 BOM 和 DOM 对象。

12.2　Object 对象

在 ECMAScript 语法规范中，Object 对象是其他一切对象的基础。虽然在实际项目开发中对 Object 对象的使用并不多，但由于其他对象都是 Object 对象继承而来的，理解一下 Object 对象的内涵还是有很大帮助的。

下面是一个使用 Object 对象的代码示例（详见源代码 ch12 目录中的 ch12-es-obj-object.html 文件）。

【代码 12-1】

```
01  <script type="text/javascript">
02      var obj = new Object();
03      obj = {
04          id : 123,
05          username : "tina",
06          email : "tina@email.com"
07      };
08      console.log(obj.hasOwnProperty("id"));
09      console.log(obj.hasOwnProperty("username"));
10      console.log(obj.hasOwnProperty("email"));
11      console.log(obj.toString());
12      console.log(obj.valueOf());
13  </script>
```

关于【代码 12-1】的分析如下：

第 02～07 行代码创建了一个对象 obj，并进行了初始化操作。

第 08～10 行代码通过使用 hasOwnProperty()方法，检测了对象 obj 中所定义的 3 个属性（id、username 和 email）是否存在，返回值为一个布尔值。Object 对象的 hasOwnProperty()方法用于判断对象是否具有某个特定的属性，且指定属性的参数必须为字符串，如 id。

第 11 行代码通过使用 toString()方法返回对象的原始字符串。在 ECMAScript 语法规范中，Object 对象没有定义这个值，因此调用该方法后的结果取决于对 ECMAScript 语法规范的具体实现。

第 12 行代码通过使用 valueOf()方法返回对象的原始值。在 ECMAScript 语法规范中，对于很多对象调用该方法后的结果与调用 toString()方法后的结果是一致的。

运行页面，控制台输出的调试信息如图 12.1 所示。

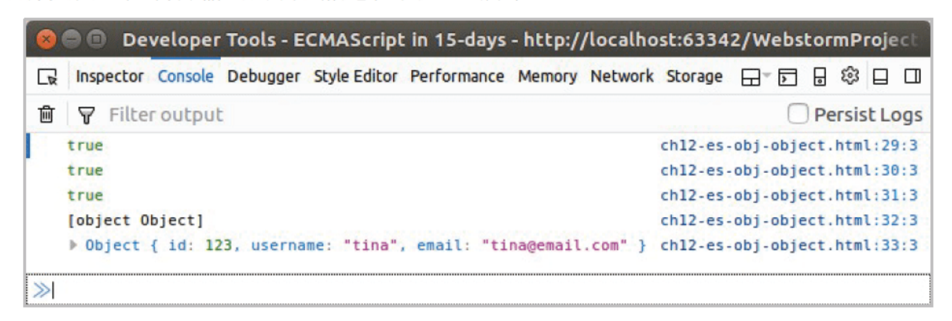


图 12.1　Object 对象

如图 12.1 所示，从第 08～10 行代码的输出结果来看，调用 hasOwnProperty()方法可以成功检测到对象 obj 中所定义的 3 个属性。

从第 11 和第 12 行代码的输出结果来看，调用 toString()方法得到的返回值是 object 类型，而调用 valueOf()方法得到的返回值是对象 obj 所定义的具体内容。

12.3 String 对象

在 ECMAScript 语法规范中，String 是用于处理字符串的对象类型。String 对象提供了大量的用于处理字符串的方法，下面详细介绍其基本使用方法。

下面是一个使用 String 对象属性和方法的代码示例（详见源代码 ch12 目录中的 ch12-es-obj-string.html 文件）。

【代码 12-2】

```
01  <script type="text/javascript">
02       var str = new String("Hello ECMAScript!"); // TODO: equals String()
03       console.log("str : " + str);
04       var str_2 = String("Hello ECMAScript!");
05       if(str == str_2) {
06            console.log("str_2 : " + str_2);
07       }
08       console.log("str length : " + str.length);          // TODO: length
09       console.log("concat() : " + str.concat(str_2));     // TODO: concat()
10       console.log("replace(/ECMA/, 'Java') : "+str.replace(/ECMA/, "Java"));
11       console.log("slice(6, 10) : " + str.slice(6, 10));      // TODO: replace()
12       console.log(str.split(" "));     // TODO: split()
13       console.log(str.split("")); // TODO: split()
14  </script>
```

关于【代码 12-2】的分析如下：

第 02 行代码通过 new 运算符定义了第一个 String 对象 str，并进行了实例化操作。

第 04 行代码直接通过 String()构造方法定义了第二个 String 对象 str_2，与第一个 String 对象 str 进行了同样的实例化操作。

第 05～07 行代码通过使用 if 条件选择语句，判断这两个 String 对象 str 和 str_2 是否相等，若相等，则通过第 06 行代码输出 String 对象 str_2 的内容。

第 08 行代码通过使用 length 属性获取了 String 对象 str 的长度。

第 09 行代码通过使用 concat()方法将两个 String 对象 str 和 str_2 的字符串内容进行连接操作。

第 10 行代码通过使用 replace()方法将 String 对象 str 中指定的内容进行字符串替换操作。

第 11 行代码通过使用 slice()方法提取 String 对象 str 中指定下标位置之间的字符串内容。

第 12 和第 13 行代码通过使用 split()方法将 String 对象 str 中的内容按照参数要求进行分割操作。

运行页面，控制台输出的调试信息如图 12.2 所示。

图 12.2　String 对象

如图 12.2 所示，从第 06 行代码的输出结果来看，第 05～07 行代码的 if 条件选择语句判断结果为真，说明两个 String 对象 str 和 str_2 是相等的。

从第 08 行代码的输出结果来看，String 对象 str 的字符串长度为 17。

从第 09 行代码的输出结果来看，concat()方法成功将两个 String 对象 str 和 str_2 的字符串内容连接到了一起。

从第 10 行代码的输出结果来看，replace()方法成功将 String 对象 str 中的“ECMA”内容替换为“Java”。

从第 11 行代码的输出结果来看，slice()方法成功提取了 String 对象 str 中指定部分的内容。

从第 12 行代码的输出结果来看，split(" ")方法（参数为带一个空格的字符串）成功将 String 对象 str 按照单词分割成数组格式。

从第 13 行代码的输出结果来看，split("")方法（参数为带一个空字符串）成功将 String 对象 str 按照字母（含空格）分割成了数组格式。

以上就是 String 对象中一些常用方法的使用过程，ECMA-262 规范中还定义了一些关于 String 对象的方法，读者可自行参看学习。

12.4　Array 对象

在 ECMAScript 语法规范中，Array 是用于处理数组的对象类型。Array 对象提供了大量的用于处理数组元素的方法，在前文中关于数组函数的内容已经介绍了一部分，本节再补充一部分。

12.4.1　Array 对象初始化

Array 对象的初始化可以通过多种方式来完成，既可以先创建后初始化，也可以创建时就进行初始化。

下面是一个创建 Array 对象和初始化方式的代码示例（详见源代码 ch12 目录中的 ch12-es-obj-array-inst.html 文件）。

【代码 12-3】

```
01   <script type="text/javascript">
```

```
02        /* inst array pattern 1 */
03        console.log("Inst Array Pattern 1 : ");
04        var arr1 = new Array();
05        arr1[0] = 1;
06        arr1[1] = 2;
07        arr1[2] = 3;
08        console.log(arr1);
09        console.log("arr1's length is : " + arr1.length);
10        console.log();
11        /* inst array pattern 2 */
12        console.log("Inst Array Pattern 2 : ");
13        var arr2 = new Array(3);
14        console.log("arr2's length is : " + arr2.length);
15        for(var i=0; i<arr2.length; i++) {
16            arr2[i] = i + 1;
17        }
18        console.log(arr2);
19        console.log();
20        /* inst array pattern 3 */
21        console.log("Inst Array Pattern 3 : ");
22        var arr3 = new Array(1, 2, 3);
23        console.log(arr3);
24        console.log("arr3's length is : " + arr3.length);
25        console.log();
26    </script>
```

关于【代码 12-3】的分析如下：

这段代码使用了 3 种方式定义并实例化了 Array 对象，分别是第 04～10 行代码、第 12～19 行代码和第 21～25 行代码。

第 04 行代码通过 new 运算符定义第一个 Array 对象 arr1。注意，该对象 arr1 没有定义参数，相当于定义了一个无固定长度的数组对象，便于后续进行动态实例化操作。

第 05～07 行代码对 Array 对象 arr1 进行了实例化操作。注意，数组起始下标是从 0 开始的，依次对 3 个数组元素进行赋值。

第 08 行代码直接通过变量名 arr1 在浏览器控制台中输出数组的内容。

第 09 行代码通过使用 length 属性在浏览器控制台中输出变量名 arr1 的数组长度。

第 13 行代码通过 new 运算符定义第二个 Array 对象 arr2。注意，该对象 arr2 定义了参数 3，相当于定义了一个固定长度为 3 的数组对象。

第 14 行代码通过使用 length 属性在浏览器控制台中输出变量 arr2 的数组长度。

第 15～17 行代码通过使用 for 循环语句对 Array 对象 arr2 进行实例化操作，赋值内容与第一个 Array 对象 arr1 相同。

第 18 行代码直接通过变量名 arr2 在浏览器控制台中输出数组的内容。

第 22 行代码通过 new 运算符定义第三个 Array 对象 arr3。注意，在定义对象 arr3 的同时直接在 Array()方法内实例化该对象，赋值内容与前两个 Array 对象 arr1 和 arr2 相同。

第 23 行代码直接通过变量名 arr3 在浏览器控制台中输出数组的内容。

第 24 行代码通过使用 length 属性在浏览器控制台中输出变量 arr3 的数组长度。

运行页面，控制台输出的调试信息如图 12.3 所示。从【代码 12-3】中的 3 种 Array 对象定义与实例化方式的输出结果来看，这 3 种方式的效果是相同的，且每种方式都有各自的优点。具体来

说，第一种方式可以创建一个无固定长度的数组对象；第二种方式可以在实例化前就获取数组对象的长度；第三种方式最简洁、快速，在实际开发中可根据代码需求择优选择。

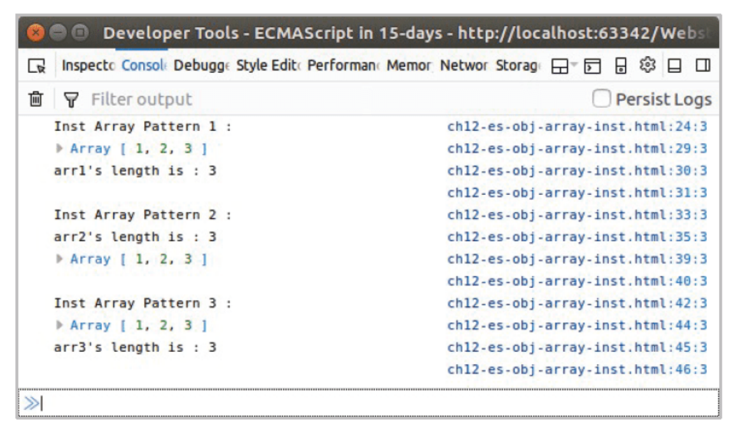


图 12.3　Array 对象定义与初始化

12.4.2　Array 对象连接操作

Array 对象定义了一个 concat()函数方法用于数组的连接操作。下面是一个使用 Array 对象中 concat()方法连接数组的代码示例（详见源代码 ch12 目录中的 ch12-es-obj-array-concat.html 文件）。

【代码 12-4】

```
01  <script type="text/javascript">
02      /* inst array 1 */
03      console.log("Inst Array 1 : ");
04      var arr1 = new Array(1, 2, 3);
05      console.log(arr1);
06      console.log();
07      console.log("arr1 concat {3, 2, 1}");
08      console.log(arr1.concat(3, 2, 1));
09      console.log();
10      console.log("Inst Array 2 : ");
11      var arr2 = new Array(4, 5, 6);
12      console.log(arr2);
13      console.log("Inst Array 3 : ");
14      var arr3 = new Array(7, 8, 9);
15      console.log(arr3);
16      console.log();
17      console.log("Array 1 concat Array 2 & 3 : ");
18      console.log(arr1.concat(arr2, arr3));
19      console.log();
20  </script>
```

关于【代码 12-4】的分析如下：

第 04 行代码通过 new 运算符定义了第一个 Array 对象 arr1，并实例化一个固定长度的数组对象（1，2，3）。

第 05 行代码直接通过变量名 arr1 在浏览器控制台中输出数组的内容。

第 08 行代码直接通过 concat()方法连接数组（3，2，1），并在浏览器控制台中输出连接操作后的结果。

第 11 行代码通过 new 运算符定义了第二个 Array 对象 arr2，并实例化一个固定长度的数组对象（4，5，6）。

第 12 行代码直接通过变量名 arr2 在浏览器控制台中输出数组的内容。

第 14 行代码通过 new 运算符定义了第三个 Array 对象 arr3，并实例化一个固定长度的数组对象（7，8，9）。

第 15 行代码直接通过变量名 arr3 在浏览器控制台中输出数组的内容。

第 18 行代码直接通过 concat()方法同时连接数组 arr2 和数组 arr3，并在浏览器控制台中输出连接操作后的结果。

运行页面，控制台输出的调试信息如图 12.4 所示。



图 12.4　Array 对象数组连接方法

如图 12.4 所示，从第 08 行代码输出的结果来看，数组 arr1 直接连接数组{3，2，1}的操作结果为{1，2，3，3，2，1}，说明直接连接数组的操作成功了。

从第 18 行代码输出的结果来看，数组 arr1 同时连接数组 arr2 和 arr3 的操作结果为{1，2，3，4，5，6，7，8，9}，说明同时连接两个数组的操作也没有问题。

通过【代码 12-4】可以看到，concat()方法既可以直接连接数组项，也可以同时连接多个数组对象。

12.4.3　Array 对象模拟堆栈

Array 对象中定义了一组用于模拟堆栈操作的方法，分别是 push()和 pop()。在 ECMA-262 规范中关于这两个方法的说明如下：

- push()方法：用于向数组的末尾添加一个或多个元素，并返回数组新的长度。
- pop()方法：用于删除并返回数组的最后一个元素。

下面是一个使用 Array 对象中 push()和 pop()模拟堆栈操作方法的代码示例（详见源代码 ch12 目录中的 ch12-es-obj-array-push-pop.html 文件）。

【代码 12-5】

```
01   <script type="text/javascript">
02        /* inst array */
03        console.log("Inst Array : ");
04        var arr = new Array(1, 2, 3);
05        console.log(arr);
06        console.log();
07        console.log("arr.push(4, 5, 6)");
08        console.log(arr.push(4, 5, 6));
09        console.log(arr);
10        console.log();
11        console.log("arr.pop()");
12        console.log(arr.pop());
13        console.log(arr);
14        console.log(arr.pop());
15        console.log(arr);
16        console.log(arr.pop());
17        console.log(arr);
18        console.log();
19   </script>
```

关于【代码 12-5】的分析如下：

第 04 行代码通过 new 运算符定义了第一个 Array 对象 arr，并实例化一个固定长度的数组对象（1，2，3）。

第 05 行代码直接通过变量名 arr 在浏览器控制台中输出数组的内容。

第 08 行代码直接通过 push()方法将数组（4，5，6）加入数组对象 arr 的末尾，并通过第 09 行代码在浏览器控制台中输出 push 操作后的结果。

第 12、第 14 和第 16 行代码分别通过调用 3 次 pop()方法取出数组对象 arr 的末尾元素，并通过第 13、第 15 和第 17 行代码在浏览器控制台中输出 pop 操作后的结果。

运行页面，控制台输出的调试信息如图 12.5 所示。从第 08 行代码输出的结果来看，push()方法返回的是数组修改完成后的长度。从第 13、第 15 和第 17 行代码输出的结果来看，pop()方法返回的是数组末尾的元素。

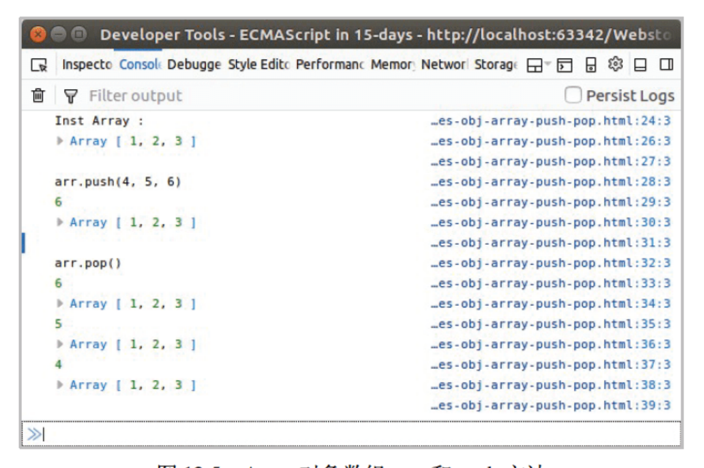


图 12.5　Array 对象数组 pop 和 push 方法

12.5 Number 对象

在 ECMAScript 语法规范中，Number 是对原始数值的包装对象。Number 对象提供了一些专用于处理原始数值的方法，在前面的章节中已经陆续介绍了一部分比较常规的方法，本节再补充几个 ECMA-262 规范中比较有特点的方法。

- toFixed()方法：用于将数字转换为字符串，结果可以保留小数点后指定位数的数字。该方法主要用于把 Number 数字按照指定的小数位数进行四舍五入操作。
- toExponential()方法：用于把 Number 数字转换为指数计数法。
- toPrecision()方法：在 Number 对象的值超出指定位数时，将其转换为指数计数法。

下面是一个使用以上几个 Number 对象方法的代码示例（详见源代码 ch12 目录中的 ch12-es-obj-number.html 文件）。

【代码 12-6】

```
01  <script type="text/javascript">
02      var d = new Number(2 / 3);
03      console.log("2/3 = " + d);
04      console.log("(2/3). toFixed(3) = " + d.toFixed(3));
05      console.log();
06      var e = new Number(1000000);
07      console.log("1000000.toExponential(1) = " + e.toExponential(1));
08      console.log();
09      var p = new Number(1000000);
10      console.log("1000000.toPrecision(3) = " + e.toPrecision(3));
11      console.log();
12  </script>
```

关于【代码 12-6】的分析如下：

这段代码分别测试了前面提到的 Number 对象的 3 个方法。

第 02 行代码通过 new 运算符定义第一个 Number 对象 d，并进行实例化操作（2 / 3），通过第 03 行代码在浏览器控制台中输出计算结果。

第 04 行代码通过 toFixed(3)方法将第一个 Number 对象 d 进行四舍五入，并保留了小数点后三位有效数字。

第 06 行代码通过 new 运算符定义第二个 Number 对象 e，并进行实例化操作（1000000）。

第 07 行代码通过 toExponential(1)方法将第二个 Number 对象 e 转换为指数计数法，参数定义为 1 表示需要保留的小数位数。

第 09 行代码通过 new 运算符定义第三个 Number 对象 p，并进行实例化操作（1000000）。

第 10 行代码通过 toPrecision(3)方法将第三个 Number 对象 p 进行在超出位数后转换为指数计数法的操作，参数定义为 3 表示有效的位数。

运行页面，控制台输出的调试信息如图 12.6 所示。

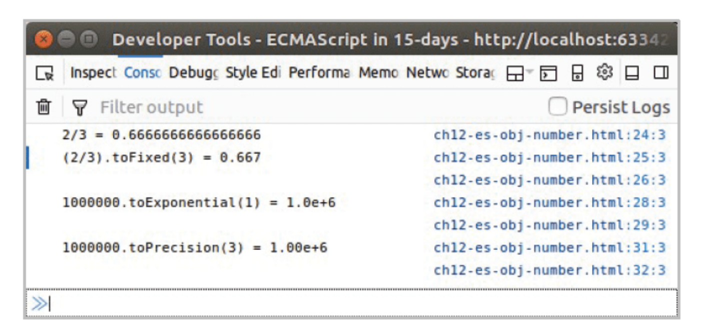


图 12.6　Number 对象方法

如图 12.6 所示，从第 03 和第 04 行代码输出的结果对比来看，toFixed(3)方法已成功将计算结果按照参数指定的位数 3，进行了四舍五入和保留小数位数的操作。

从第 07 行代码输出的结果来看，toExponential(1)方法已成功将数字 1000000 按照指数计数法进行了转换。

从第 10 行代码输出的结果来看，toPrecision(3)方法已成功将数字 1000000 按照参数定义有效位数 3，进行了转换为指数计数法的操作。

12.6　Boolean 对象

在 ECMAScript 语法规范中，Boolean 对象只有两个值，即 true 与 false。在使用 Boolean 对象时，有构造函数和转换函数两种方式，它们在 ECMA-262 规范中的具体说明如下：

- 构造函数方式：通过在 Boolean()函数前使用运算符 new，将参数转换为一个布尔值，并返回一个包含该值的 Boolean 对象。
- 转换函数方式：在 Boolean()函数前不使用运算符 new，将参数转换为一个原始的布尔值，并返回该值。

下面是一个使用 Boolean 对象的代码示例（详见源代码 ch12 目录中的 ch12-es-obj-boolean.html 文件）。

【代码 12-7】

```
01  <script type="text/javascript">
02      var bTrue = new Boolean(true);
03      console.log("true to string is " + bTrue.toString());
04      var bOne = Boolean(1);
05      console.log("Boolean(1) to string is " + bOne.toString());
06      console.log();
07      var bFalse = new Boolean(false);
08      console.log("false to string is " + bFalse.toString());
09      var bZero = Boolean(0);
10      console.log("Boolean(0) to string is " + bZero.toString());
11      var bNull = Boolean();
12      console.log("Boolean() to string is " + bNull.toString());
```

```
13        console.log();
14    </script>
```

关于【代码 12-7】的分析如下：

这段代码分别测试了前面提到的 Boolean 对象的构造函数和转换函数的使用方法。

第 02 行代码通过 new 运算符定义第一个 Boolean 对象 bTrue，并进行实例化操作（true），通过第 03 行代码使用 toString()方法在浏览器控制台中输出结果。

第 04 行代码通过 new 运算符定义第二个 Boolean 对象 bOne，并进行实例化操作（1），通过第 05 行代码使用 toString()方法在浏览器控制台中输出结果。

第 07 行代码通过 new 运算符定义第三个 Boolean 对象 bFalse，并进行实例化操作（false），通过第 08 行代码使用 toString()方法在浏览器控制台中输出结果。

第 09 行代码通过 new 运算符定义第四个 Boolean 对象 bZreo，并进行实例化操作（0），通过第 10 行代码使用 toString()方法在浏览器控制台中输出结果。

第 11 行代码通过 new 运算符定义第五个 Boolean 对象 bNull，注意该方法在实例化时并没有参数，通过第 12 行代码使用 toString()方法在浏览器控制台中输出结果。

运行页面，控制台输出的调试信息如图 12.7 所示。

如图 12.7 所示，从第 03 和第 08 行代码输出的结果对比来看，使用运算符 new 的构造函数方式可以成功创建布尔值 true 和 false 的 Boolean 对象。

从第 05 和第 10 行代码输出的结果对比来看，不使用运算符 new 的转换函数方式同样可以得到布尔值 true 和 false。

另外，从第 12 行代码的输出结果来看，使用不带参数的 Boolean()函数时得到的返回值是 false，这一点需要读者多加注意。

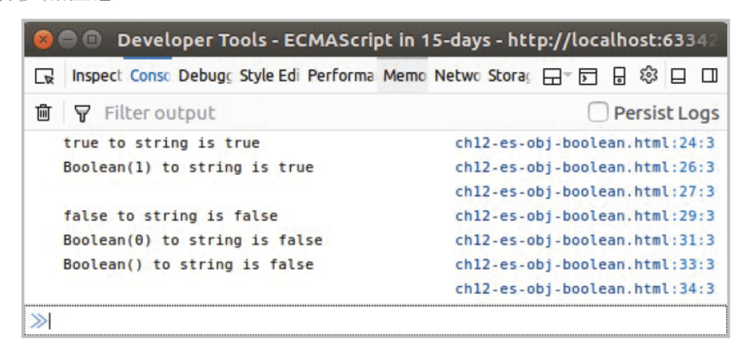


图 12.7　Boolean 对象方法

12.7　Date 对象

在 ECMAScript 语法规范中，Date 对象是用于处理日期和时间的包装类型，通过应用 Date 对象可以很方便地操作日期和时间。

12.7.1　Date 对象基础

Date 对象提供了一系列专用于处理日期和时间的方法，在前面章节中已经介绍了几个比较常用的方法，本小节再补充几个 ECMA-262 规范中提供的常用方法。

- setFullYear(year, month, day)方法：用于设置年份。具体参数和返回值的描述如下：
  - year：该参数是必需的，表示年份的四位整数，用本地时间来表示。
  - month：该参数是可选的，表示月份的数值（0～11），用本地时间来表示。
  - day：该参数是可选的，表示月中某一天的数值（1～31），用本地时间来表示。
  - 返回值：返回被调整过的日期的毫秒数。
- setMonth(month, day)方法：用于设置月份。具体参数描述如下：
  - month：该参数是必需的，表示月份的数值介于 0（一月）～11（十二月）之间。
  - day：该参数是可选的，表示月中某一天的数值（1～31），用本地时间来表示。
  - 返回值：返回被调整过的日期的毫秒数。
- setDate(day)方法：用于设置月中的某一天。具体参数和返回值的描述如下：
  - day：该参数是必需的，表示月中某一天的数值（1～31）。
  - 返回值：返回被调整过的日期的毫秒数。
- setHours(hour, min, sec, millisec)方法：用于设置时间中的小时字段。具体参数和返回值的描述如下：
  - hour：该参数是必需的，表示小时的数值，介于 0（午夜）~23（晚上 11 点）之间，用本地时间来表示。
  - min：该参数是可选的，表示分钟的数值（0～59），用本地时间来表示。
  - sec：该参数是可选的，表示秒的数值（0～59），用本地时间来表示。
  - millisec：该参数是可选的，表示毫秒的数值（0～999），用本地时间来表示。
  - 返回值：返回被调整过的日期的毫秒数。
- setMinutes(min, sec, millisec)方法：用于设置时间中的分钟字段。具体参数和返回值的描述如下：
  - min：该参数是必需的，表示分钟的数值（0～59），用本地时间来表示。
  - sec：该参数是可选的，表示秒的数值（0～59），用本地时间来表示。
  - millisec：该参数是可选的，表示毫秒的数值（0～999），用本地时间来表示。
  - 返回值：返回被调整过的日期的毫秒数。
- setSeconds(sec, millisec)方法：用于设置时间中的秒数字段。具体参数和返回值的描述如下：
  - sec：该参数是必需的，表示秒数的数值（0～59），用本地时间来表示。
  - millisec：该参数是可选的，表示毫秒的数值（0～999），用本地时间来表示。
  - 返回值：返回被调整过的日期的毫秒数。
- setMilliseconds(millisec)方法：用于设置时间中的毫秒数字段。具体参数和返回值的描述如下：
  - millisec：该参数是必需的，表示毫秒数的数值（0～999），用本地时间来表示。
  - 返回值：返回被调整过的日期的毫秒数。

- setTime(millisec)方法：通过毫秒数设置 Date 对象。具体参数和返回值的描述如下：
  - millisec：该参数是必需的，表示要设置的日期和时间距离 GMT 时间 1970 年 1 月 1 日 0 点之间的毫秒数。
  - 返回值：返回被设置的毫秒数。
- toLocaleString()方法：该方法可根据本地时间把 Date 对象转换为字符串，并返回结果。

12.7.2 Date 对象应用（一）

通过上一小节介绍的 Date 对象的几个方法，可以对年、月、日及时间等参数进行读取和设置。下面是一个使用前面提到的 Date 对象方法的代码示例（详见源代码 ch12 目录中的 ch12-es-obj-date.html 文件）。

【代码 12-8】

```
01  <script type="text/javascript">
02      var date = new Date();
03      date.setFullYear(2008);
04      console.log("setFullYear(2008) = ");
05      console.log(date);
06      date.setFullYear(2008, 7, 8);
07      console.log("setFullYear(2008,7,8) = ");
08      console.log(date);
09      date.setMonth(0);
10      console.log("setMonth(0) = ");
11      console.log(date);
12      date.setMonth(0, 1);
13      console.log("setMonth(0,1) = ");
14      console.log(date);
15      date.setDate(15);
16      console.log("setData(15) = ");
17      console.log(date);
18  </script>
```

关于【代码 12-8】的分析如下：

这段代码分别测试了前面提到的 Date 对象的几个方法。

第 02 行代码通过 new 运算符定义一个 Date 对象 date。

第 03 行代码通过调用 setFullYear(2008)方法将时间设置到 2008 年。

第 06 行代码通过调用 setFullYear(2008, 7, 8)方法将时间设置到 2008 年 8 月 8 日。

第 09 行代码通过调用 setMonth(0)方法将时间设置到 1 月份。

第 12 行代码通过调用 setMonth(0, 1)方法将时间设置到 1 月 1 日。

第 15 行代码通过调用 setDate(15)方法将时间设置到 15 日。

运行页面，控制台输出的调试信息如图 12.8 所示。

如图 12.8 所示，从第 05 行代码输出的结果来看，setFullYear(2008)方法成功将日期的年份设置为 2008 年。注意，因为第 03 行代码调用的 setFullYear()方法仅定义了一个年份的参数，所以月份和天数保留了系统当前的日期。

从第 08 行代码输出的结果来看，setFullYear(2008,7,8)方法成功将日期设置为 2008 年 8 月 8

日。因为第 06 行代码调用的 setFullYear()方法定义了全部年月日的参数，所以日期全部被修改成功了。

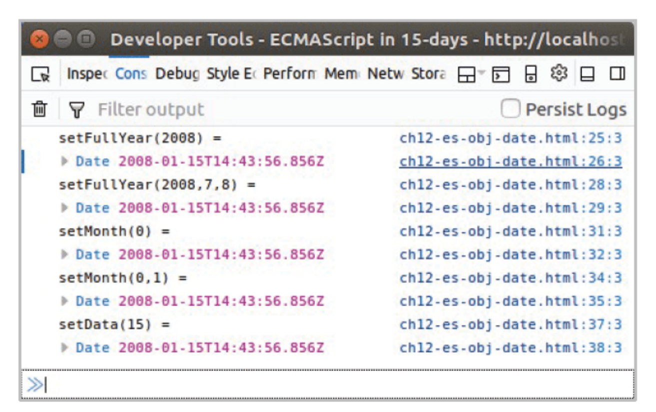


图 12.8　Date 对象应用（1）

从第 11 行代码输出的结果来看，setMonth(0)方法成功将日期的月份设置为 2008 年 1 月 8 日，而年份和天数仍保留第 06 行代码设置的日期。

从第 14 行代码输出的结果来看，setMonth(0,1)方法成功将日期的月份设置为 2008 年 1 月 1 日，而年份仍保留第 06 行代码设置的日期。

从第 17 行代码输出的结果来看，setDate(15)方法成功将日期的天数设置为 2008 年 1 月 15 日，而年份和月份仍保留第 14 行代码设置的日期。

12.7.3　Date 对象应用（二）

下面再看一个使用另外几个 Date 对象方法的代码示例（详见源代码 ch12 目录中的 ch12-es-obj-date-time.html 文件）。

【代码 12-9】

```
01  <script type="text/javascript">
02      var date = new Date();
03      console.log("Current Time :");
04      console.log(date.toLocaleString());
05      console.log();
06      console.log("setHours(8) :");
07      date.setHours(8);
08      console.log(date.toLocaleString());
09      console.log("setHours(8,8,8) :");
10      date.setHours(8, 8, 8);
11      console.log(date.toLocaleString());
12      console.log("setMinutes(30) :");
13      date.setMinutes(30);
14      console.log(date.toLocaleString());
15      console.log("setMinutes(30,30) :");
16      date.setMinutes(30,30);
17      console.log(date.toLocaleString());
18      console.log("setSeconds(0) :");
```

```
19        date.setSeconds(0);
20        console.log(date.toLocaleString());
21        console.log("setTime(1234567890) :");
22        date.setTime(1234567890);
23        console.log(date.toLocaleString());
24    </script>
```

关于【代码 12-9】的分析如下：

这段代码分别测试了前面提到的 Date 对象的另外几个方法。

第 02 行代码通过 new 运算符定义一个 Date 对象 date，此时对象 date 相当于获取了当前时间。

第 04 行代码通过调用 toLocaleString()方法，将对象 date 的当前时间转化为本地时间格式，并在浏览器控制台中进行输出。

第 07 行代码通过调用 setHours(8)方法将小时数设置为 8 时。

第 10 行代码通过调用 setHours(8, 8, 8)方法将时间数设置为 8 时 8 分 8 秒。

第 13 行代码通过调用 setMinutes(30)方法将分钟数设置为 30 分。

第 16 行代码通过调用 setMinutes(30, 30)方法将分钟数和秒数设置为 30 分 30 秒。

第 19 行代码通过调用 setSeconds(0)方法将秒数设置为 0 秒。

第 22 行代码通过调用 setTime(1234567890)方法，将时间设置为距 1970 年 1 月 1 日 0 时 0 分 0 秒往后 1234567890 毫秒数的时间。

运行页面，控制台输出的调试信息如图 12.9 所示。

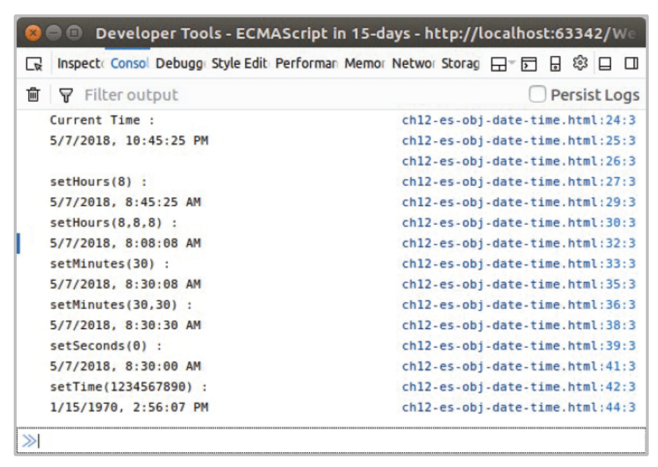


图 12.9　Date 对象应用（2）

如图 12.9 所示，从第 04 行代码输出的结果来看，toLocaleString()方法成功将当前时间转换为本地时间格式。

从第 08 行代码输出的结果来看，setHours(8)方法成功将小时数设置为 8 时，同时年、月、日、分钟和秒数都保持原始数值。

从第 11 行代码输出的结果来看，setHours(8, 8, 8)方法成功将小时数、分钟数和秒数设置为 8 时 8 分 8 秒。

从第 14 行代码输出的结果来看，setMinutes(30)方法成功将分钟数设置为 30 分，而小时数和秒数都保留第 10 行代码的设置。

从第 17 行代码输出的结果来看，setMinutes(30, 30)方法成功将分钟数和秒数设置为 30 分 30 秒，而小时数仍保留第 10 行代码的设置。

从第 20 行代码输出的结果来看，setSeconds(0)方法成功将秒数设置为 0 秒，而小时数和分钟分别沿用了第 10 行和第 16 行代码的设置。

从第 23 行代码输出的结果来看，通过 setTime(1234567890)方法成功将时间设置为距 1970 年 1 月 1 日 0 时 0 分 0 秒往后 1234567890 毫秒数的时间（具体时间是 1970/1/15 下午 2:56:07）。

12.8 本章小结

本章主要介绍了 ECMAScript 语法规范中关于对象分类的知识，包括基础的 Object 对象和常规的 String、Number、Array、Boolean 和 Date 对象等方面的内容，并通过一些具体实例进行了讲解。

第 13 章

对象新特性

在 ECMAScript 6 版本语法规范中为对象增加了许多新特性，如对象属性的简洁表示法、符号对象 Symbol、集合对象 Set 和 Map 等，本章将介绍这些新特性及其使用方法。

13.1 对象属性的简洁表示法

ECMAScript 6 语法规范为对象属性新增了一种简洁表示法，通过这种简洁的定义方式，可以为创建对象和函数返回值提供更快捷的操作。

下面是一个使用对象属性简洁表示法的代码示例（详见源代码 ch13 目录中的 ch13-es-obj-new-prop.html 文件）。

【代码 13-1】

```
01  <script type="text/javascript">
02      var username = "tina";
03      var email = "tina@email.com";
04      var obj = {
05          id : 123,
06          username,
07          email,
08          printInfo() {
09              console.log("id : " + this.id);
10            console.log("username : " + this.username);
11            console.log("email : " + this.email);
12          }
13      };
14      console.log("------ print obj ------");
15      obj.printInfo();
16  </script>
```

关于【代码 13-1】的分析如下：

这段代码主要就是定义了一个对象，包括 3 个属性和一个函数。其中，属性和函数的定义均采用了简洁表示法。

第 06 行代码定义一个属性 username。注意，这里并没有定义属性值，但第 02 行代码定义了同名的变量 username 并进行初始化赋值"tina"，此时属性 username 的值就取自于同名变量 username 的值。

第 07 行代码定义的属性 email 与第 06 行代码定义的属性 username 类似。

第 08～12 行代码定义一个函数 printInfo()，用于输出属性信息。注意，这里同样使用了没有关键字 function 来定义的简洁表示法。

运行页面，控制台输出的调试信息如图 13.1 所示。通过对象属性简洁表示法定义的对象，与传统方式定义的对象功能上完全一致，同时可以提高代码的编写效率。

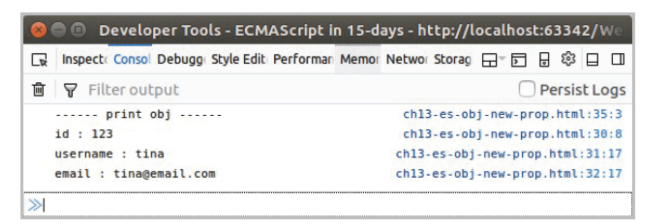


图 13.1　对象属性简洁表示法（1）

下面继续看一个使用对象属性简洁表示法定义函数返回值的代码示例（详见源代码 ch13 目录的中 ch13-es-obj-new-prop-return.html 文件）。

【代码 13-2】

```
01   <script type="text/javascript">
02       function getUserinfo() {
03           var id = "123";
04           var userinfo = "tina";
05           var email = "tina@email.com";
06           return {id, userinfo, email};
07       }
08      var info = getUserinfo();
09       console.log(info);
10   </script>
```

关于【代码 13-2】的分析如下：

这段代码主要就是定义了一个函数，其中包括 3 个变量。第 06 行代码的返回值使用了对象属性的简洁表示法。

运行页面，控制台输出的调试信息如图 13.2 所示。通过对象属性简洁表示法可以简化函数返回值的定义，效率提高还是非常显著的。

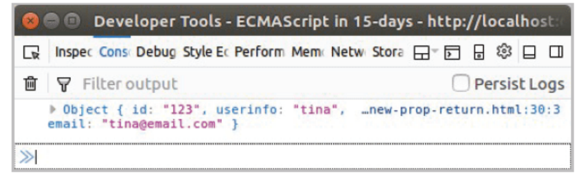


图 13.2　对象属性简洁表示法（2）

13.2　Symbol 数据类型

符号对象 Symbol 是 ECMAScript 6 语法规范中新增的数据类型，通过 Symbol 可以创建一个唯一的值，而该特性使得 Symbol 非常适合作为标识符来使用。下面通过几个简单的代码实例介绍符号对象 Symbol 的使用方法。

13.2.1　定义 Symbol 对象

定义 Symbol 对象的方法是通过函数 Symbol()来生成的。先看一段使用函数 Symbol()生成 Symbol 对象的代码（详见源代码 ch13 目录中的 ch13-es-symbol-define.html 文件）。

【代码 13-3】

```
01  <script type="text/javascript">
02      console.log("var s = Symbol();");
03      var s = Symbol();
04      console.log("typeof s : " + typeof s);
05  </script>
```

关于【代码 13-3】的分析如下：

第 03 行代码通过函数 Symbol()定义了一个 Symbol 对象 s。

第 04 行代码通过运算符 typeof 检查对象 s 的类型，并将结果输出到浏览器控制台中。

运行页面，控制台输出的调试信息如图 13.3 所示。从浏览器控制台的输出结果来看，对象 s 的类型是 Symbol，而不是字符串等其他类型。

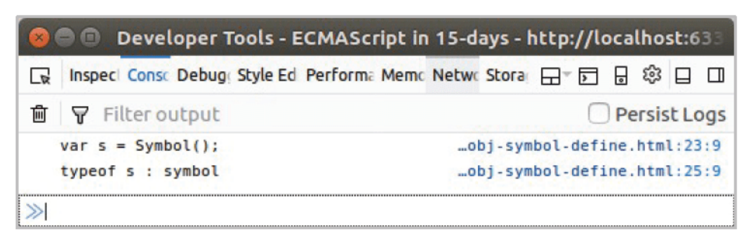


图 13.3　定义 Symbol 对象

13.2.2　Symbol 对象的唯一性

ECMAScript 语法规范中规定了凡是通过函数 Symbol()生成的 Symbol 对象均是唯一的，或者说是独一无二的。这个概念该如何理解呢？

下面是一个表明 Symbol 对象具有唯一性的代码示例（详见源代码 ch13 目录中的 ch13-es-symbol-unique.html 文件）。

【代码 13-4】

```
01  <script type="text/javascript">
```

```
02        console.log("var s1 = Symbol();");
03        var s1 = Symbol();
04        console.log("var s2 = Symbol();");
05        var s2 = Symbol();
06        if (s1 === s2)
07            console.log("s1 === s2 ? true");
08        else
09            console.log("s1 === s2 ? false");
10    </script>
```

关于【代码 13-4】的分析如下：

第 03 行和第 05 行代码同时通过函数 Symbol()定义了两个 Symbol 对象 s1 和 s2。

第 06～09 行代码通过 if 条件选择语句判断对象 s1 和对象 s2 是否严格相等，并将结果输出到浏览器控制台中。

运行页面，控制台输出的调试信息如图 13.4 所示。虽然根据【代码 13-4】的定义可以看到对象 s1 和对象 s2 均是 Symbol 数据类型，但从浏览器控制台的输出结果来看，对象 s1 和对象 s2 是非严格相等的。

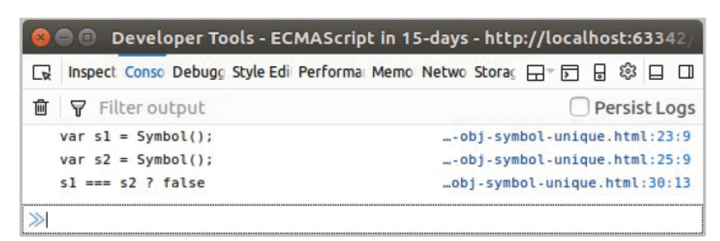


图 13.4　Symbol 对象唯一性

这个结果对于常规 ECMAScript 对象是无法得到的，但对于 ECMAScript 6 新增的 Symbol 数据类型就是如此。在 ECMAScript 语法规范中，如此设计 Symbol 数据类型自然有其目的，通过 Symbol 就可以实现对象自定义属性的唯一性了。

13.2.3　Symbol 定义属性名

通过前文的介绍我们知道了 Symbol 对象的唯一性，可以借助这个特性为对象定义属性名。因为通过 Symbol 数据类型定义的属性名都是独一无二的，所以也就可以保证自定义属性名不会产生冲突，这一点在大型的项目开发中非常有用。

下面是一个通过 Symbol 定义属性名的代码示例（详见源代码 ch13 目录中的 ch13-es-symbol-prop-name.html 文件）。

【代码 13-5】

```
01    <script type="text/javascript">
02        console.log("var s = Symbol();");
03        var s = Symbol();
04        var obj = {};
05        obj[s] = "Symbol DataType";
06        console.log("obj[s] : " + obj[s]);
07        var obj2 = {
```

```
08            [s]: "Symbol DataType 2"
09        };
10        console.log("obj2[s] : " + obj2[s]);
11    </script>
```

关于【代码 13-5】的分析如下：

第 03 行代码通过函数 Symbol()定义了一个 Symbol 对象 s。

第 04 和第 05 行代码是一种定义对象属性名的方式；第 07～09 行代码是另一种定义对象属性名的方式。这两种方式都是可行的，设计人员可自行选择。

运行页面，控制台输出的调试信息如图 13.5 所示。从浏览器控制台的输出结果来看，两种定义对象属性名的方式都得到了正确的结果。

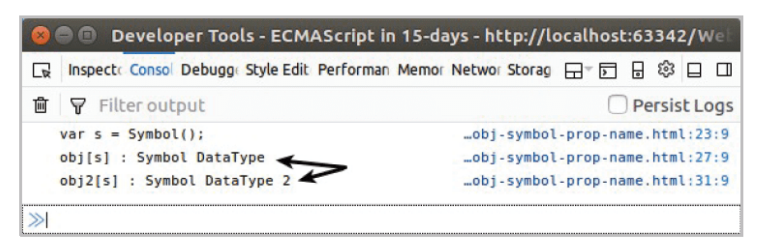


图 13.5　Symbol 定义属性名

13.3　Set 数据类型

ECMAScript 6 语法规范中新增了一个集合类的数据类型——Set 类型，该类型是一种包含无重复元素的有序列表，这点与传统数组类型是不同的。

13.3.1　定义和遍历 Set 数据类型

定义 Set 数据类型的方式与定义传统 ECMAScript 对象的方式基本一致，即通过运算符 new 来生成一个 Set 类型对象，并进行初始化操作。

下面是一个定义 Set 数据类型的代码示例（详见源代码 ch13 目录中的 ch13-es-obj-set-define.html 文件）。

【代码 13-6】

```
01    <script type="text/javascript">
02        var set = new Set();
03        set.add(1);
04        set.add("a");
05        set.add(2);
06        set.add("b");
07        set.add(3);
08        set.add("c");
09        for (let s of set) {
10            console.log(s);
```

```
11        }
12    </script>
```

关于【代码 13-6】的分析如下：

第 02 行代码通过 new 运算符定义了一个 Set 数据类型对象 set。

第 03～08 行代码通过 Set 数据类型的 add()方法对对象 set 进行初始化操作。注意，初始化的数据类型同时包括整数和字符类型。

第 09～11 行代码通过 for…of…循环迭代语句将对象 set 的内容输出到浏览器控制台中。

运行页面，控制台输出的调试信息如图 13.6 所示。从浏览器控制台的输出结果来看，通过 for…of…循环迭代语句可以遍历 Set 类型对象。

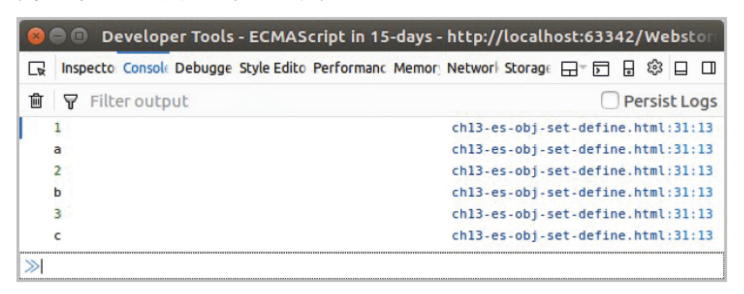


图 13.6　定义 Set 数据类型

13.3.2　判断 Set 集合中的值

ECMAScript 语法规范中为 Set 数据类型提供了一个 has()方法，可以用来判断一个值是否包含在 Set 集合中。

下面是判断一个值是否包含在 Set 集合中的代码示例（详见源代码 ch13 目录中的 ch13-es-obj-set-has.html 文件）。

【代码 13-7】

```
01    <script type="text/javascript">
02        var set = new Set();
03        set.add(1);
04        set.add("a");
05        set.add(2);
06        set.add("b");
07        set.add(3);
08        set.add("c");
09        for (let i = 1; i <= 3; i++) {
10            if (set.has(i)) {
11                console.log("set has " + i + ".");
12            }
13        }
14    </script>
```

关于【代码 13-7】的分析如下：

第 09～13 行代码通过 for 循环语句定义了一个自变量 i 从数值 1~3 的循环。

第 10 行代码通过 Set 数据类型的 has()方法，判断 for 循环的自变量 i 的取值是否包含在 Set

对象中。

运行页面，控制台输出的调试信息如图 13.7 所示。通过 Set 数据类型的 has()方法成功判断出数值是否包含在集合中。

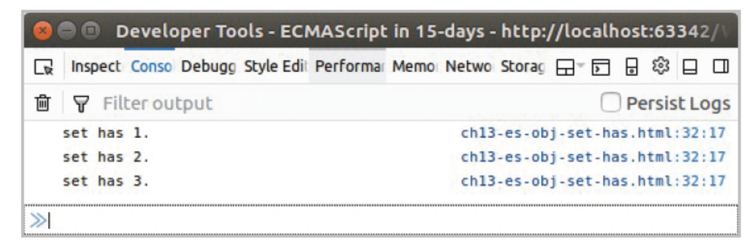


图 13.7　判断数值是否包含在 Set 集合中

13.3.3　删除和清空 Set 集合

ECMAScript 语法规范中还为 Set 数据类型提供了删除 delete()方法和清空 clear()方法，分别用来删除 Set 集合中的某个值和清空 Set 集合中的全部值。

下面是一个删除和清空 Set 集合中数据的代码示例（详见源代码 ch13 目录中的 ch13-es-obj-set-del-clear.html 文件）。

【代码 13-8】

```
01  <script type="text/javascript">
02      var set = new Set();
03      set.add(1);
04      set.add("a");
05      set.add(2);
06      set.add("b");
07      set.add(3);
08      set.add("c");
09      console.log("--- init set ---");
10      printSet(set);
11      for (let i = 1; i <= 3; i++) {
12          if (set.has(i)) {
13              set.delete(i);
14          }
15      }
16      console.log("--- delete set ---");
17      printSet(set);
18      set.clear();
19      console.log("--- clear set ---");
20      printSet(set);
21      /**
22       * func --- printSet
23       * @param set
24       */
25      function printSet(set) {
26          var sSet = "set : ";
27          for (let s of set) {
28              sSet += s + "  ";
```

```
29          }
30          console.log(sSet);
31      }
32  </script>
```

关于【代码 13-8】的分析如下：

第 11～15 行代码通过 for 循环语句定义了一个自变量 i 从数值 1~3 的循环。其中，第 12 行代码通过 Set 数据类型的 has()方法，判断 for 循环的自变量 i 的取值是否包含在 Set 对象中；第 13 行代码通过 Set 数据类型的 delete()方法，依次删除每个包含在 Set 对象中的数据。

第 18 行代码通过 Set 数据类型的 clear()方法，在执行第 13 行代码删除 Set 集合中的数据之后，再次清空 Set 对象中全部剩余的数据。

运行页面，控制台输出的调试信息如图 13.8 所示。通过 Set 数据类型的 delete()方法和 clear()方法成功删除和清空了集合中的数据。

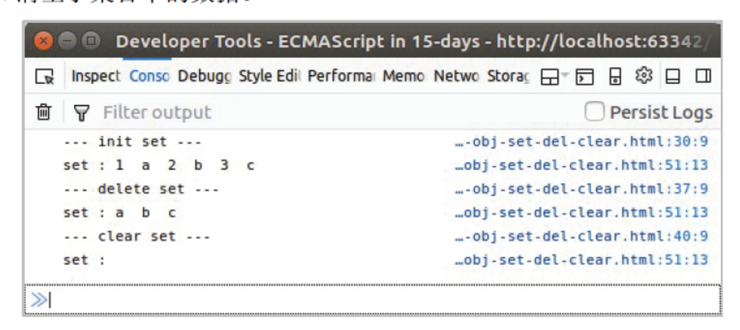


图 13.8　删除和清空 Set 集合中的值

13.4　Map 数据类型

ECMAScript 6 语法规范中还新增了另一个集合类的数据类型——Map 类型，该类型是一种包含有序键值对的有序列表，其中键和值可以是任何类型的对象。

13.4.1　定义 Map 数据类型和基本存取操作

定义 Map 数据类型的方式与定义传统 ECMAScript 对象的方式基本一致，即通过运算符 new 生成一个 Map 类型对象，然后分别使用 set()方法和 get()方法进行存储键值对及键取值的操作。

下面是一个定义 Map 数据类型和进行基本存取操作的代码示例（详见源代码 ch13 目录中的 ch13-es-obj-map-basic.html 文件）。

【代码 13-9】

```
01  <script type="text/javascript">
02      var map = new Map();
03      map.set(1, "a");
04      map.set(2, "b");
05      map.set(3, "c");
```

```
06        console.log("map's size : " + map.size);
07        for (let i = 1; i <= 3; i++) {
08            console.log("map(" + i + ", " + map.get(i) + ")");
09        }
10    </script>
```

关于【代码 13-9】的分析如下：

第 02 行代码通过 new 运算符定义了一个 Map 数据类型对象 map。

第 03～05 行代码通过 Map 数据类型的 set()方法对对象 map 进行初始化操作，一共存储了三组键值对。

第 06 行代码通过 Map 数据类型的 size 属性获取集合的长度。

第 07～09 行代码是在 for 循环语句中使用 Map 数据类型的 get()方法，通过对象 map 的键获取对应的值。

运行页面，控制台输出的调试信息如图 13.9 所示。从浏览器控制台的输出结果来看，Map 对象是通过键来调用值的。

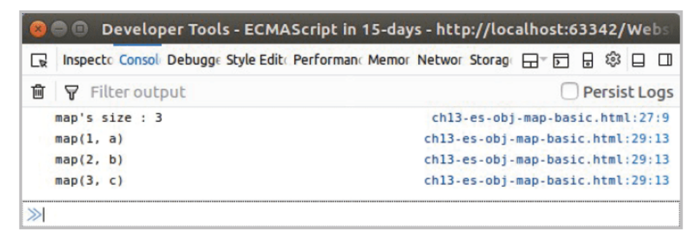


图 13.9　定义 Map 数据类型

13.4.2　判断 Map 集合中的值

ECMAScript 语法规范中同样为 Map 数据类型提供了一个 has()方法，用来判断一个键值对是否包含在 Map 集合中。

下面是判断一个键值对是否包含在 Map 集合中的代码示例（详见源代码 ch13 目录中的 ch13-es-obj-map-has.html 文件）。

【代码 13-10】

```
01    <script type="text/javascript">
02        var map = new Map();
03        map.set(1, "a");
04        map.set(2, "b");
05        map.set(3, "c");
06        console.log("map's size : " + map.size);
07        for (let i = 1; i <= map.size; i++) {
08            if (map.has(i))
09                console.log("map(" + i + ", " + map.get(i) + ")");
10        }
11    </script>
```

关于【代码 13-10】的分析如下：

第 08 行代码在 if 语句中通过使用 Map 数据类型的 has()方法判断 for 循环的自变量 i 的取值作

为键是否包含在 Map 对象中。

运行页面，控制台输出的调试信息如图 13.10 所示。从浏览器控制台的输出结果来看，通过 Map 数据类型的 has()方法成功判断出键值对是否包含在集合中。

图 13.10　判断键值对是否包含在 Map 集合中

13.4.3　删除和清空 Map 集合

ECMAScript 语法规范中还为 Map 数据类型提供了删除 delete()方法和清空 clear()方法，分别用来删除 Map 集合中的某个键值对和清空 Map 集合中的全部键值对。

下面是一个删除和清空 Map 集合中键值对的代码示例（详见源代码 ch13 目录中的 ch13-es-obj-map-del-clear.html 文件）。

【代码 13-11】

```
01  <script type="text/javascript">
02      var map = new Map();
03      map.set(1, "a");
04      map.set(2, "b");
05      map.set(3, "c");
06      console.log("map's size : " + map.size);
07      for (let i = 1; i <= 3; i++) {
08          console.log("map(" + i + ", " + map.get(i) + ")");
09      }
10      console.log("delete map(2, 'b') : " + map.delete(2));
11      console.log("map's size : " + map.size);
12      for (let i = 1; i <= 3; i++) {
13          if (map.has(i))
14              console.log("map(" + i + ", " + map.get(i) + ")");
15      }
16      console.log("clear map : " + map.clear());
17      console.log("map's size : " + map.size);
18      for (let i = 1; i <= 3; i++) {
19          if (map.has(i))
20              console.log("map(" + i + ", " + map.get(i) + ")");
21      }
22  </script>
```

关于【代码 13-11】的分析如下：

第 10 行代码通过 Map 数据类型的 delete()方法删除了 Map 对象中的第二组键值对数据。

第 16 行代码通过 Map 数据类型的 clear()方法，在执行第 10 行代码删除数据之后，再次清空 Map 对象中全部剩余的数据。

运行页面，控制台输出的调试信息如图 13.11 所示。从浏览器控制台的输出结果来看，通过 Map 数据类型的 delete()方法和 clear()方法成功删除和清空集合中的键值对。

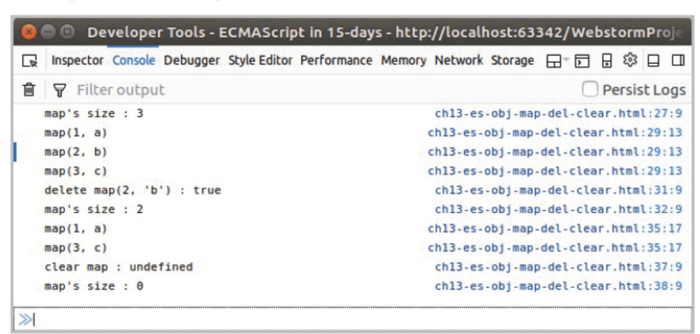


图 13.11　删除和清空 Map 集合中的键值对

13.5　本章小结

本章主要介绍了 ECMAScript 6 语法规范中关于对象新特性的知识，包括 ECMAScript 对象属性简洁表示法、Set 对象与 Map 对象的基本概念和使用方法，并通过一些具体实例进行了讲解。

第 14 章

正则表达式

本章将介绍 ECMAScript 正则表达式相关的内容。在 ECMAScript 语法规范中，正则表达式是通过一个 RegExp 对象来定义的，也就是说 ECMAScript 语法规范将正则表达式按照对象模式进行设计。

14.1　正则表达式基础

14.1.1　什么是正则表达式

正则表达式（Regular Expression）主要用来查询、检索或替换所有符合某个模式（规则）的文本。因此，正则表达式其实就是一种文本模式，用于描述在搜索文本时要匹配的一个或多个字符串。

一般来说，正则表达式是由普通字符（a～z、A～Z 等）、数字（0～9）及一些特殊字符（"\" "|" "*" "^" "$" "?" "!"等）组合而成的一种逻辑公式，其表示具有一定规则的字符串，并通过规则来达到过滤文本的目的。

在计算机编程领域，正则表达式已被绝大多数高级程序设计语言（如 C、Java、PHP、Perl 和 JavaScript 等）所采纳。由于正则表达式在程序设计中的重要性，因此很多高级程序设计语言专门为正则表达式功能实现了开发库，将每项具体功能整合在一起，极大地方便了程序员的测试和调用。

14.1.2　RegExp 对象语法

在 ECMAScript 语法规范中，为 RegExp 对象的使用定义了具体规则。如果打算使用正则表达式，那首先要定义一个 RegExp 对象。

1. 通过 new 运算符定义 RegExp 对象

```
var regExp = new RegExp(pattern, attributes);
```

其中，变量 regExp 保存了 RegExp()构造方法的返回值，表示一个 RegExp 对象。参数 pattern 表示模式（正则表达式模式或具体正则表达式），即当使用该 RegExp 对象在一个字符串中检索时，需要匹配的就是参数 pattern。参数 attributes 表示一个可选的属性值："g" "i"和"m"，这 3 个属性值分别表示指定全局匹配、区分大小写的匹配和多行匹配。另外，参数 attributes 是 ECMAScript 标准化之后的产物，标准化之前是不支持"m"属性的。若参数 pattern 是正则表达式，而不是字符串，则必须省略该参数 attributes。

2. 不通过 new 运算符定义 RegExp 对象

```
var regExp = RegExp(pattern, attributes);
```

如果不使用 new 运算符，而是将 RegExp()作为函数调用，那么该方式与使用 new 运算符定义 RegExp 对象的方式是一样的。需要注意的是，当参数 pattern 是正则表达式时，函数只返回模式 pattern，而不会创建一个新的 RegExp 对象。

3. 直接量

ECMAScript 规范还支持使用直接量来定义模式字符串，具体格式如下：

```
/pattern/attributes
```

其中，参数 pattern 和 attributes 的含义与使用 RegExp 对象定义模式字符串时的参数含义是一样的。

4. 异常处理

如果参数 pattern 是不合法的正则表达式，或者参数 attributes 不是"g" "i"和"m"这 3 个合法字符，就会抛出异常；如果参数 pattern 是合法的 RegExp 对象，却没有省略参数 attributes，那也会抛出异常。因此，设计人员在使用正则表达式时需要注意这两点，避免出现错误。

14.1.3 RegExp 对象模式

ECMAScript 语法规范为 RegExp 对象的使用定义了具体规则方法，包括 RegExp 对象的属性和方法、方括号、元字符、量词、修饰符等。下面就介绍 RegExp 对象的规则方法。

1. RegExp 对象方法

ECMAScript 语法规范为 RegExp 对象一共定义了 3 个主要方法，具体见表 14.1。

表 14.1 RegExp 对象方法

方法名称	描述	返回值	示例
test	检索字符串中指定的值	true 或 false	regExp.test(pattern)
exec	检索字符串中指定的值	返回检索到的值及其位置	regExp exec(pattern)
compile	编译正则表达式		regExp.compile (pattern)

2. RegExp 对象修饰符标记

ECMAScript 语法规范为 RegExp 对象定义了 3 个修饰符标记，用于匹配检索模式，具体见表 14.2。

表 14.2　RegExp 对象修饰符标记

修饰符标记名称	描述
g	执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）
i	执行对大小写不敏感的匹配
m	执行多行匹配

3. RegExp 对象属性

ECMAScript 语法规范为 RegExp 对象定义了一组属性，具体见表 14.3。

表 14.3　RegExp 对象属性

属性名称	描述
global	检查 RegExp 对象是否具有修饰符标记“g”
ignoreCase	检查 RegExp 对象是否具有修饰符标记“i”
multiline	检查 RegExp 对象是否具有修饰符标记“m”
lastIndex	标示开始下一次匹配的字符起始位置（整数数值）
source	正则表达式的源文本

4. 方括号

在 ECMAScript 规范的 RegExp 对象中，方括号“[]”用于查找某个范围内的字符，具体见表 14.4。

表 14.4　RegExp 对象方括号表达式

表达式	描述
[abc]	检索方括号集合之内的任何字符
[^abc]	检索任何不在方括号集合之内的字符
[0-9]	查找任何从 0~9 的数字
[a-z]	查找任何从小写 a 到小写 z 的字符
[A-Z]	查找任何从大写 A 到大写 Z 的字符
[A-z]	查找任何从大写 A 到小写 z 的字符（依据 ASCII 编码的顺序）

5. 元字符

在 ECMAScript 规范的 RegExp 对象中，元字符用于查找具有特殊含义的字符，具体见表 14.5。

表 14.5　RegExp 对象元字符

表达式	描述
\w	查找单词字符（具体就是子母、数字或下画线）
\W	查找非单词字符（与\w 相对应）
\d	查找数字
\D	查找非数字字符（与\d 相对应）

（续表）

表达式	描述
\s	查找空白字符
\S	查找非空白字符（与\s 相对应）
\b	匹配单词边界
\B	匹配非单词边界（与\b 相对应）
.	小数点符号用于查找单个字符（除换行符和行结束符）
\n	查找换行符
\r	查找回车符
\t	查找制表符
\xxx	查找以八进制数 xxx 规定的字符
\xdd	查找以十六进制数 dd 规定的字符
\uxxxx	查找以十六进制数 xxxx 规定的 Unicode 字符

备　注
正则表达式里“单词”的含义就是连续不少于一个的“\w”，而非传统意义的英文单词。

6. 量词

在 ECMAScript 规范的 RegExp 对象中，量词用于查找具有重复定义的字符，具体见表 14.6。

表 14.6　RegExp 对象量词

表达式	描述
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复 n 次
{n,}	重复 n 次或更多次
{n,m}	重复 n 到 m 次
^	匹配字符串的开始
$	匹配字符串的结束

7. 分组

在 ECMAScript 规范的 RegExp 对象中，分组通过小括号“(...)”来定义，可以实现对一组多个字符重复次数的查找。

8. 分枝

在 ECMAScript 规范的 RegExp 对象中，分枝通过符号“|”来定义。具体来说，分枝就是在同时存在几种规则的情况下，只要能满足其中任意一种规则就表示匹配成功，从逻辑上讲分枝与“或”的关系是一致的。

14.2　RegExp 对象方法

本节将介绍如何使用正则表达式 RegExp 对象的方法，具体包括 test()方法、exec()方法和 compile() 方法。

14.2.1　test 方法

test()方法用于检测一个字符串是否匹配某个模式，该方法会根据检索结果返回一个布尔值。下面是一个使用 test()方法的简单代码示例（详见源代码 ch14 目录中的 ch14-es-regexp-test.html 文件）。

【代码 14-1】

```
01  <script type="text/javascript">
02      var strTxt = "Hello ECMAScript!";
03      console.log("Searched Txt : Hello ECMAScript!");
04      var regExp01 = new RegExp("ECMAScript");
05      var result01 = regExp01.test(strTxt);
06      console.log("test 'ECMAScript' return : " + result01);
07      var regExp02 = new RegExp("ECMAScript");
08      var result02 = regExp02.test(strTxt);
09      console.log("test 'ECMAScript' return : " + result02);
10      var regExp03 = new RegExp(" ");
11      var result03 = regExp03.test(strTxt);
12      console.log("test blank return : " + result03);
13  </script>
```

关于【代码 14-1】的分析如下：

第 02 行代码定义了一个字符串变量 strTxt，并初始化为 "Hello ECMAScript!"，用作被检索的字符串。

第 04 行代码通过 new 关键字创建了 RegExp 对象的第一个实例 regExp01，模式参数定义为字符串 "ECMAScript"。

第 05 行代码通过使用 test()方法，判断第 04 行代码定义的模式字符串 "ECMAScript"，能否在第 02 行代码定义的字符串变量 strTxt 中检索出来，检索结果的返回值保存在变量 result01 中。

第 07 行代码通过 new 关键字创建了 RegExp 对象的第二个实例 regExp02，模式参数定义为字符串 "ECMAScript"。注意，这里将大写字母换成了小写字母。

第 08 行代码通过使用 test()方法，判断第 07 行代码定义的模式字符串 "ECMAScript"，能否在第 02 行代码定义的字符串变量 strTxt 中检索出来，结果返回值保存在变量 result02 中。

第 10 行代码通过 new 关键字创建了 RegExp 对象的第三个实例 regExp03，模式参数定义为空格（" "）字符串。

第 11 行代码通过使用 test()方法，判断第 10 行代码定义的模式字符串（空格），能否在第 02 行代码定义的字符串变量 strTxt 中检索出来，检索结果的返回值保存在变量 result03 中。

运行页面，控制台输出的调试信息如图 14.1 所示。

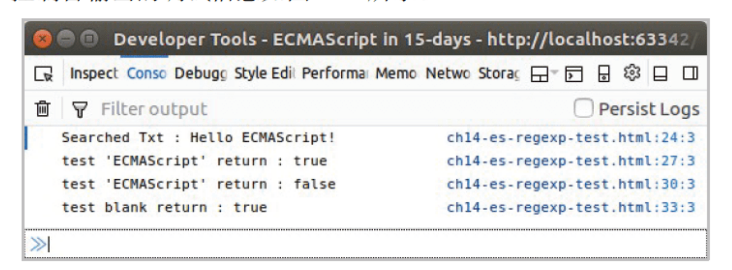


图 14.1　RegExp 对象 test()方法

如图 14.1 所示，从第 06 行代码输出的结果来看，第 04 行代码定义的模式字符串 "ECMAScript" 被成功检索到了，返回结果为 true。

从第 09 行代码输出的结果来看，第 07 行代码定义的模式字符串 "ECMAScript"没有被检索到，返回结果为 false。这说明使用 RegExp 对象的 test()方法检索时，字母默认是区分大小写的。

从第 12 行代码输出的结果来看，第 10 行代码定义的模式字符串空格同样被成功检索到了，返回结果为 true。这说明 RegExp 对象是将空格当作字符串来处理的，这样设计无疑是合理的，因为像空格这样的特殊字符在文本中是经常用到的。

14.2.2　exec 方法

与 test()方法类似，exec()方法同样用于检测一个字符串是否匹配某个模式。二者的主要区别是返回值不同，test()方法的返回值为布尔值，而 exec()方法的返回值为检索到的内容。

下面是一个使用 exec()方法的代码示例（详见源代码 ch14 目录中的 ch14-es-regexp-exec.html 文件），这段代码是在【代码 14-1】基础上修改而成的。

【代码 14-2】

```
01  <script type="text/javascript">
02      var strTxt = "Hello ECMAScript!";
03      console.log("Searched Txt : Hello ECMAScript!");
04      var regExp01 = new RegExp("ECMAScript");
05      var result01 = regExp01.exec(strTxt);
06      console.log("exec 'ECMAScript' return : " + result01);
07      var regExp02 = new RegExp("ECMAScript");
08      var result02 = regExp02.exec(strTxt);
09      console.log("exec 'ECMAScript' return : " + result02);
10      var regExp03 = new RegExp("e");
11      var result03 = regExp03.exec(strTxt);
12      console.log("exec 'e' return : " + result03);
13  </script>
```

关于【代码 14-2】的分析如下：

第 02 行代码定义了一个字符串变量 strTxt，并初始化为 "Hello ECMAScript!"，用作被检索的字符串。

第 04 行代码通过 new 关键字创建了 RegExp 对象的第一个实例 regExp01，模式参数定义为字

符串 "ECMAScript"。

第 05 行代码通过使用 exec()方法，检索第 04 行代码定义的模式字符串 "ECMAScript"，是否包括在第 02 行代码定义的字符串变量 strTxt 中，并将检索结果的返回值保存在变量 result01 中。

第 07 行代码通过 new 关键字创建了 RegExp 对象的第二个实例 regExp02，模式参数定义为字符串“ECMAScript”。注意，这里将大写字母换成了小写字母。

第 08 行代码通过使用 exec()方法，检索第 07 行代码定义的模式字符串 "ECMAScript"，是否包括在第 02 行代码定义的字符串变量 strTxt 中，并将检索结果的返回值保存在变量 result02 中。

第 10 行代码通过 new 关键字创建了 RegExp 对象的第三个实例 regExp03，模式参数定义为字母 "e" 的字符串。

第 11 行代码通过使用 exec()方法，检索第 10 行代码定义的模式字符串 "e"，是否包括在第 02 行代码定义的字符串变量 strTxt 中，并将检索结果的返回值保存在变量 result03 中。

运行页面，控制台输出的调试信息如图 14.2 所示。

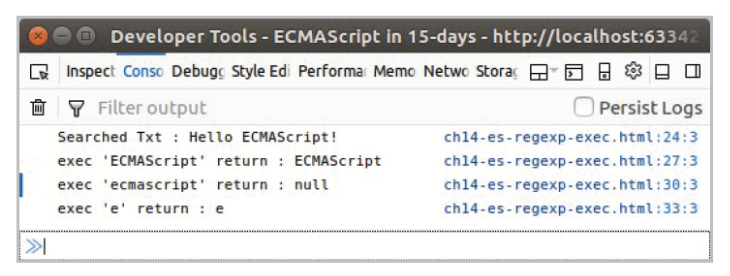


图 14.2　RegExp 对象 exec()方法

如图 4.2 所示，从第 06 行代码输出的结果来看，第 04 行代码定义的模式字符串 "ECMAScript" 被成功检索到了，返回值为 "ECMAScript"。

从第 09 行代码输出的结果来看，第 07 行代码定义的模式字符串 "ECMAScript" 没有被检索到，返回值为空对象 null。这说明 RegExp 对象的 exec()方法与 test()方法一样，在检索字母时默认是区分大小写的。

从第 12 行代码输出的结果来看，第 10 行代码定义的模式字符串 "e" 被成功检索到了，返回值为字符串 "e"，再次印证了 exec()方法是区分字母大小写的。

14.2.3　compile 方法

compile()方法用于在脚本执行过程中编译正则表达式，也可以用于改变和重新编译正则表达式。这个方法可能不如前两个方法常用，但也是一个非常重要的 RegExp 对象方法。

下面是一个使用 compile()方法的代码示例（详见源代码 ch14 目录中的 ch14-es-regexp-compile.html 文件）。

【代码 14-3】

```
01  <script type="text/javascript">
02      var strTxt = "Hello ECMAScript!";
03      console.log("Searched Txt : Hello ECMAScript!");
04      var regExp01 = new RegExp("ECMAScript");
```

```
05        var result01 = regExp01.test(strTxt);
06        console.log("test 'ECMAScript' return : " + result01);
07        regExp01.compile("ECMAScript");
08        console.log("compile 'ECMAScript' changes to 'ECMAScript'.");
09        var result02 = regExp01.test(strTxt);
10        console.log("test 'ECMAScript' return : " + result02);
11    </script>
```

关于【代码 14-3】的分析如下：

第 02 行代码定义了一个字符串变量 strTxt，并初始化为 "Hello ECMAScript!"，用作被检索的字符串。

第 04 行代码通过 new 关键字创建了 RegExp 对象的第一个实例 regExp01，模式参数定义为字符串 "ECMAScript"。

第 05 行代码通过使用 test()方法，判断第 04 行代码定义的模式字符串 "ECMAScript"，是否包括在第 02 行代码定义的字符串变量 strTxt 中，并将检索结果的返回值保存在变量 result01 中。

第 07 行代码对变量 regExp01 使用 compile()方法，通过将参数定义为字符串 "ECMAScript" 重新定义了模式字符串，即将大写字母换成小写字母。

第 09 行代码再次通过使用 test()方法，检索第 07 行代码中重新定义的模式字符串 "ECMAScript"，是否包括在第 02 行代码定义的字符串变量 strTxt 中，并将检索结果的返回值保存在变量 result02 中。

运行页面，控制台输出的调试信息如图 14.3 所示。

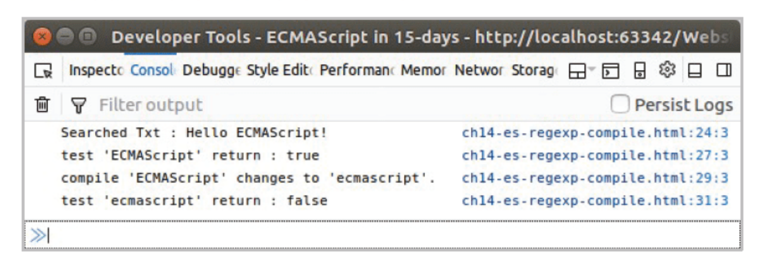


图 14.3　RegExp 对象 compile 方法

如图 14.3 所示，从第 06 行代码输出的结果来看，第 04 行代码定义的模式字符串 "ECMAScript" 被成功检索到了，返回结果为 true。

从第 09 行代码输出的结果来看，第 07 行代码重新定义的模式字符串 "ECMAScript" 没有被检索到，返回结果为 false。这说明第 07 行代码中重新定义的模式字符串被 compile()方法编译成功了，导致第 09 行代码中再次使用 test()方法检索时没有搜到字符串 "ECMAScript"。

14.3　RegExp 对象修饰符标记

本节将介绍如何使用正则表达式 RegExp 对象的修饰符标记，主要包括 g 和 i 两个修饰符标记。

14.3.1　“g”修饰符标记

“g”修饰符标记用于定义正则表达式对象是否执行全局检索。

下面是一个使用“g”修饰符标记的简单代码示例（详见源代码 ch14 目录中的 ch14-js-regexp-g.html 文件）。

【代码 14-4】

```
01  <script type="text/javascript">
02      var strTxt = "ECMAScript ECMAScript ECMAScript";
03      console.log("Searched Txt : ECMAScript ECMAScript ECMAScript");
04      var regExp = /ECMA/;
05      for (var i = 0; i < 3; i++) {
06          var result = regExp.exec(strTxt);
07          console.log("exec /ECMA/ return : " + result + " at " + regExp.lastIndex);
08      }
09      var regExp_g = /ECMA/g;
10      for (var j = 0; j < 3; j++) {
11          var result_g = regExp_g.exec(strTxt);
12          console.log("exec /ECMA/g return : " + result_g + " at " + regExp_g.lastIndex);
13      }
14  </script>
```

关于【代码 14-4】的分析如下：

第 02 行代码定义了一个字符串变量 strTxt，并初始化为 "ECMAScript ECMAScript ECMAScript"，用作被检索的字符串。

第 04 行代码通过直接量方式定义了 RegExp 对象的第一个实例 regExp，模式参数定义为字符串“/ECMA/”。

第 05～08 行代码通过一个 for 循环语句，使用 exec()方法检索第 04 行代码定义的模式字符串“/ECMA/”是否包括在第 02 行代码定义的字符串变量 strTxt 中，并将检索结果的返回值保存在变量 result 中。其中，第 07 行代码使用 RegExp 对象的 lastIndex 属性获取了下一次检索位置的整数值，该属性会在后面详细介绍。

第 09 行代码再次通过直接量方式定义了 RegExp 对象的第二个实例 regExp_g，模式参数定义为字符串“/ECMA/g”。注意，这里添加了“g”修饰符标记。

第 10～13 行代码再次通过一个 for 循环语句，使用 exec()方法检索第 09 行代码定义的模式字符串“/ECMA/g”是否包括在第 02 行代码定义的字符串变量 strTxt 中，并将检索结果的返回值保存在变量 result_g 中。其中，第 12 行代码再次使用 RegExp 对象的 "lastIndex" 属性获取了下一次检索位置的整数值。

运行页面，控制台输出的调试信息如图 14.4 所示。

如图 14.4 所示，从第 07 行代码输出的结果来看，第 04 行代码定义的模式字符串“/ECMA/”被成功检索到了，返回值为“ECMA”，位置为 0。如图 14.4 中箭头所指（重复 3 次），第 07 行代码输出的结果重复了完全相同的 3 次，说明每次执行完第 06 行代码的 exec()方法后，下一次检索的位置又重新回归到初始位置 0。

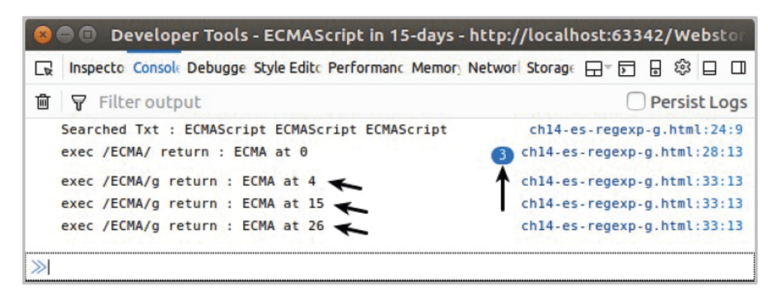


图 14.4　RegExp 对象“g”修饰符标记

从第 12 行代码输出的结果来看，第 09 行代码定义的模式字符串“/ECMA/g”被成功检索到了，返回值为“ECMA”，且 3 次调用返回的位置值均不同（如图 14.4 中箭头所指：4、15 和 26），说明每次执行完第 11 行代码的 exec()方法后，下一次检索的位置是按照顺序计算的。由此可见，第 09 行代码定义的“g”修饰符标记起到了作用。

14.3.2　“i”修饰符标记

“i”修饰符标记用于定义正则表达式对象执行大小写不敏感的检索。

下面是一个使用“i”修饰符标记的简单代码示例（详见源代码 ch14 目录中的 ch14-es-regexp-i.html 文件）。

【代码 14-5】

```
01  <script type="text/javascript">
02        var strTxt = "Hello ECMAScript!";
03        console.log("Searched Txt : Hello ECMAScript!");
04        var regExp = /E/;
05        var result = regExp.exec(strTxt);
06        console.log("exec /E/ return : " + result);
07        var regExp_i = /E/i;
08        var result_i = regExp_i.exec(strTxt);
09        console.log("exec /E/i return : " + result_i);
10  </script>
```

关于【代码 14-5】的分析如下：

第 02 行代码定义了一个字符串变量 strTxt，并初始化为 "Hello ECMAScript!"，用作被检索的字符串。

第 04 行代码通过直接量方式定义了 RegExp 对象的第一个实例 regExp，模式参数定义为字符串“/E/”。注意，这里定义的模式字符串是默认区分大小写的。

第 05 行代码通过使用 exec()方法检索第 04 行代码定义的模式字符串“/E/”是否包括在第 02 行代码定义的字符串变量 strTxt 中，并将检索结果的返回值保存在变量 result 中。

第 07 行代码通过直接量方式定义了 RegExp 对象的第二个实例 regExp_i，模式参数定义为字符串“/E/i”。注意，这里定义的模式字符串通过使用“i”修饰符标记，表示对大小写是不敏感的。

第 08 行代码通过使用 exec()方法检索第 07 行代码定义的模式字符串“/E/i”是否包括在第 02 行代码定义的字符串变量 strTxt 中，并将检索结果的返回值保存在变量 result_i 中。

运行页面，控制台输出的调试信息如图 14.5 所示。

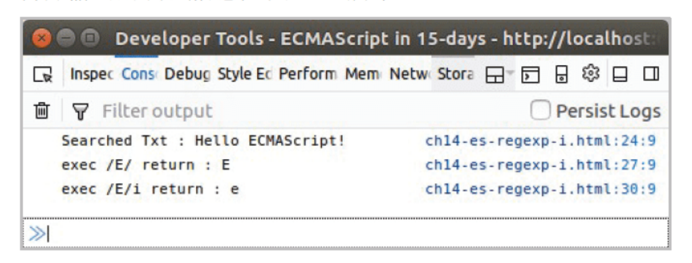


图 14.5　RegExp 对象"i"修饰符标记

如图 14.5 所示，从第 06 行代码输出的结果来看，第 04 行代码定义的模式字符串“/E/”被成功检索到了，返回值为“E”。

从第 09 行代码输出的结果来看，第 07 行代码定义的模式字符串“/E/i”被成功检索到了，返回值为“e”。注意，检索到的返回值是小写字母“e”，说明第 09 行代码定义的“i”修饰符标记起到了作用。

14.3.3　“g”和“i”修饰符标记组合

“g”和“i”这两个修饰符标记可以组合在一起使用，用于定义正则表达式对象执行全局且大小写不敏感的检索。

下面是一个使用“g”和“i”组合修饰符标记的简单代码示例（详见源代码 ch14 目录中的 ch014-es-regexp-gi.html 文件）。

【代码 14-6】

```
01  <script type="text/javascript">
02      var strTxt = "ECMAScript ECMAScript ECMAScript";
03      console.log("Searched Txt : ECMAScript ECMAScript ECMAScript");
04      var regExp_gi = /ECMAS/gi;
05      for (var i = 0; i < 3; i++) {
06          var result_gi = regExp_gi.exec(strTxt);
07          console.log("exec /ECMA/gi return : " + result_gi + " at " + regExp_gi.lastIndex);
08      }
09  </script>
```

关于【代码 14-6】的分析如下：

第 02 行代码定义了一个字符串变量 strTxt，并初始化为 "ECMAScript ECMAScript ECMAScript"，用作被检索的字符串。

第 04 行代码通过直接量方式定义了 RegExp 对象的第一个实例 regExp_gi，模式参数定义为字符串“/ECMAS/gi”。注意，这里同时使用了“g”和“i”组合修饰符标记。

第 05～08 行代码在 for 循环语句块中使用 exec()方法检索第 04 行代码定义的模式字符串“/ECMAS/gi”是否包括在第 02 行代码定义的字符串变量 strTxt 中，并将检索结果的返回值保存在变量 result_gi 中。其中，第 07 行代码使用了 RegExp 对象的“lastIndex”属性获取了下一次检索位置的整数值。

运行页面，控制台输出的调试信息如图 14.6 所示。

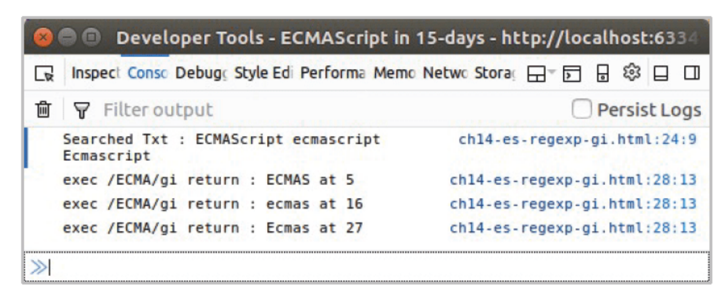


图 14.6 RegExp 对象“g”和“i”修饰符标记组合

如图 14.6 所示，从第 07 行代码输出的结果来看，第 04 行代码定义的模式字符串“/ECMAS/gi”被成功检索到了，且 3 次调用的返回值均不同（分别为“ECMAS”“ecmas”和“Ecmas”），返回的位置值也均不同（分别为 5、16 和 27），说明每次执行完第 06 行代码的 exec()方法后，下一次检索的位置是按照顺序计算的。由此可见，第 04 行代码定义的“g”和“i”修饰符标记组合起到了作用。

14.4 本章小结

本章主要介绍了 ECMAScript 语法规范中关于正则表达式 RegExp 对象的知识，具体包括正则表达式基础、RegExp 对象方法、RegExp 对象修饰符标记等方面的内容，并通过一些具体实例进行了讲解。

第 15 章

面向对象编程

本章将介绍 ECMAScript 语法规范中面向对象编程的内容，具体包括面向对象、对象作用域、继承机制，以及 ECMAScript 6 面向对象新特性等内容。可以说，面向对象编程是 ECMAScript 语法规范中学习难度比较大，但也是最能提高设计人员编程技术的部分。

15.1 面向对象基础

15.1.1 什么是“面向对象”

“面向对象编程（OOP）”是一个非常时髦的词汇，大家在讨论高级程序设计语言时，免不了要提到该程序设计语言是否支持“面向对象”的技术。

为什么面向对象如此重要呢？那是因为早期程序设计语言都是“面向过程”的，非常适用于早期的应用程序（代码量小且可控）。后来，随着应用程序需要实现的功能越来越多及设计难度的越来越高，程序的代码量也开始呈几何级数增长，维护难度越来越大，这时候“面向过程”的程序设计语言就显得落后了。

人类面对困难总是知难而上的。聪明的程序员开始思考如何对代码进行优化、复用和重构，从根本上解决“面向过程”设计理念的不足。于是，“面向对象”的设计理念就开始出现了，高级程序设计语言也如雨后春笋般，呈现出蓬勃的发展趋势。这里比较有代表性就是 C++语言和 Java 语言，也是目前大学的必修课。“面向对象”的设计理念解决了很多“面向过程”设计理念的不足，从根本上解决了代码优化、复用和重构的问题，可以说是计算机软件发展过程中重要的里程碑。

ECMAScript 脚本语言作为后起之秀，自然也会在设计理念上保持先进性，因此也实现了 ECMAScript 版本的“面向对象”功能。当然，ECMAScript 的“面向对象”技术与传统的“面向对象”技术在实现上还是有一定区别的（很多程序员认为 ECMAScript 的实现更为先进，这里就不

做深究了），但笔者认为有区别不重要，关键是能够实现设计理念就是成功的。

15.1.2 面向对象的特点

一般来说，面向对象的程序设计语言需要满足以下 4 种基本功能：

- 封装：能够实现将数据或方法存储在对象中的功能。
- 聚集：能够实现将一个对象存储在另一个对象内的功能。
- 继承：能够实现将另一个类（或多个类）的属性和方法完整获取过来的功能。
- 多态：能够实现以多种方法运行的函数或方法的功能。

因为 ECMAScript 语法规范中完全支持以上这些功能要求，所以 JavaScript 被认为是“面向对象”的高级程序设计语言。

15.1.3 面向对象的专业术语

面向对象作为一种高级程序语言设计技术，自然会有其专用的技术术语。下面就列举几个常用的术语。

1. 类（class）

每个对象都由类（class）来定义，可以把类看作对象的装配工具。类（class）不仅要定义对象的接口（interface，开发者访问的属性和方法），还要定义对象的内部工作（使属性和方法发挥作用的代码）。

2. 对象实例

关于对象的实例在前面有过介绍，其实“类”与“对象”是两个紧密关联的概念。程序在使用类创建对象时，生成的对象可以称为“类的实例”或“对象实例”。

3. 类与对象的关系

这里需要特别解释的就是，在 ECMAScript 脚本语言中关于“类”与“对象”的关系。其实，在早期 ECMAScript 版本（具体说是 ECMAScript 6 之前）语法规范中并没有“类”的概念，它是通过“对象”来替代“类”的功能的。但是，在当前的 ECMAScript 6 版本（ECMA-262 规范）中，又增加了 class 关键字用来实现“类”的新特性，这一点确实让设计人员有点崩溃了。

无论怎么说，ECMAScript 脚本语言的面向对象编程与传统语言（C 或 Java 语言等）的面向对象编程还是有一定区别的。本质上，“对象”与“类”只是名称的区别，通过 ECMAScript 脚本语言提供的特性来实现面向对象编程的功能才是最终目标。

15.2　ECMAScript 对象作用域

本节将介绍 ECMAScript 对象作用域的知识，包括变量作用域和 this 关键字。

15.2.1　变量作用域

学习过 C++或 Java 等面向对象编程语言的读者都知道，类的成员属性和方法（也可以理解为对象的）都是有作用域这个概念的。具体的作用域大致可分为公有、私有和受保护 3 种形式。当然，可能每种面向对象编程语言所规定的名称或定义的形式略有区别，大体模式却是一样的。

但是，在 ECMAScript 语法规范中并没有公有、私有和受保护这几种作用域的概念。ECMAScript 语法规范中只定义了公有作用域这个形式，所有对象的属性和方法都是公有的，原则上都可以被访问和使用。

这对于大多数设计人员来说非常不习惯，于是有些设计人员就人为地规定了一些作用域的形式，但这些规定都不是标准规范，仅仅是约定俗成的，原则上是无法改变 ECMAScript 对象是公有的这个概念的。

15.2.2　this 关键字

学习 ECMAScript 脚本语言的过程中，大多数读者可能不会在意前面提到的关于作用域的概念。但是，this 关键字确实是必须掌握的、非常重要的一个概念。要理解并掌握 this 关键字，首先要大致理解作用域的概念，最关键的是要熟练掌握作用域在对象中的使用方法。

如果用一句话来概括 this 关键字的概念，那么可以将 this 关键字定义为“指向所调用方法或属性的对象”。this 关键字在具体代码应用中非常灵活，读者需要多加练习才可以熟练掌握。

下面是一个使用 this 关键字的简单代码示例（详见源代码 ch15 目录中的 ch15-es-oop-this.html 文件）。

【代码 15-1】

```
01  <script type="text/javascript">
02        console.log("------ object's this ------");
03        var oThis = new Object;
04        oThis.prop = "prop";
05        oThis.showProp = function () {
06            console.log("this.prop : " + this.prop);
07            console.log("oThis.prop : " + oThis.prop);
08        }
09        oThis.showProp();
10        console.log("------ this of the page level ------");
11        console.log("this : " + this);
12  </script>
```

关于【代码 15-1】的分析如下：

第 03 行代码通过 new 关键字创建了 Object 对象的一个对象实例，并将其存储到变量 oThis 中。

第 04 行代码为对象实例 oThis 定义了一个 prop 属性，并进行初始化操作。

第 05～08 行代码为对象实例 oThis 定义了一个 showProp 方法。其中，第 06 行代码使用 this 关键字调用了 prop 属性，并输出其属性值；第 07 行代码直接使用对象实例 oThis 调用了 prop 属性，并输出其属性值。

第 09 行代码通过对象实例 oThis 调用 showProp 方法。

第 11 行代码直接在浏览器控制台中输出 this 关键字的内容。

运行页面，控制台输出的调试信息如图 15.1 所示。通过 this 关键字调用 prop 属性和直接调用对象实例 oThis 的属性的结果是相同的。如果在页面中直接使用 this 关键字，那么 this 代表的是 Window 窗口对象。

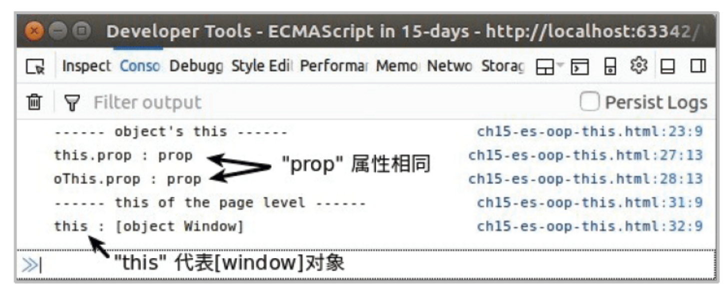


图 15.1　this 关键字

上面的代码实例介绍的就是 this 关键字的一些基本使用方法，后面的代码实例中还会有大量对 this 关键字的使用，读者可以循序渐进地学习。

15.3　创建 ECMAScript 类与对象

本节将继续介绍如何创建 ECMAScript 类与对象，包括工厂模式、构造方法方式和原型方式等方面的内容。

15.3.1　工厂模式创建类与对象

在 ECMAScript 语法规范中是没有“class”这个关键字的，也就是说我们是无法像 C 或 Java 语言那样创建“类”的。听起来有些不可思议，不使用“class”怎么定义“类”呢，但这也正是 JavaScript 脚本语言的特点。

因为 ECMAScript 语法规范中规定了一切皆为“对象”，那是不是可以用 Object 对象来定义“类”呢？答案是肯定的，就是使用设计模式中非常有名的工厂（Factory）模式。

下面是一个定义 ECMAScript 类（对象）的基本代码示例（详见源代码 ch15 目录中的 ch15-es-oop-factory-object.html 文件）。

【代码 15-2】

```
01  <script type="text/javascript">
02      var userInfo = new Object;
03      userInfo.id = "123";
04      userInfo.name = "tina";
05      userInfo.email = "tina@email.com";
06      userInfo.showInfo = function () {
07         console.log("id : " + this.id);
08         console.log("name : " + this.name);
09         console.log("email : " + this.email);
10      };
11      userInfo.showInfo();
12      userInfo = null;
13  </script>
```

关于【代码 15-2】的分析如下：

第 02 行代码通过 new 关键字创建 Object 对象的一个对象实例，并将其存储到变量 userInfo 中。

第 03～05 行代码为对象实例 userInfo 定义一组属性，并进行初始化操作。

第 06～10 行代码为对象实例 userInfo 定义一个 showInfo 方法。其中，第 07～09 行代码依次使用 this 关键字调用了对象实例 userInfo 的一组属性，并在浏览器控制台中输出其属性值。

第 11 行代码通过对象实例 userInfo 调用 showInfo 方法。

第 12 行代码以手动方式清除第 02 行代码定义的对象实例 userInfo。

运行页面，控制台输出的调试信息如图 15.2 所示。代码中定义的 userInfo 对象已经具有“类”的基本特性了，包括属性和方法的使用功能。

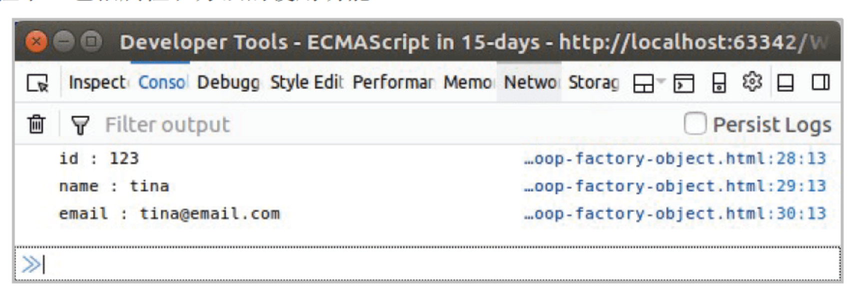


图 15.2　工厂模式创建类和对象

当然，【代码 15-2】创建“类”的方式也有很多问题，最大的问题就是 userInfo 对象是根本无法重用的，只有通过重复创建新的、类似 userInfo 对象实例的方式才可以完成“类”的功能。

15.3.2　封装的工厂模式创建类与对象

在 ECMAScript 规范下如何实现“类”的重用呢？可以在工厂模式的基础上通过封装的形式实现“类”的重用。所谓“封装”，就是通过 ECMAScript 函数的方式将定义“类”的过程封装进去，然后通过返回对象来实现创建“类”的功能。

下面是一个定义封装形式的 ECMAScript 类（对象）的代码示例（详见源代码 ch15 目录中的 ch15-es-oop-factory-func.html 文件）。

【代码 15-3】

```
01  <script type="text/javascript">
02      function createUserInfo() {
03          var userInfo = new Object;
04          userInfo.id = "123";
05          userInfo.name = "tina";
06          userInfo.email = "tina@email.com";
07          userInfo.showInfo = function () {
08              console.log("id : " + this.id);
09              console.log("name : " + this.name);
10              console.log("email : " + this.email);
11          };
12          return userInfo;
13      }
14      console.log("------ object instance 1 ------");
15      var v_UI_1 = createUserInfo();
16      v_UI_1.showInfo();
17      console.log("------ object instance 2 ------");
18      var v_UI_2 = createUserInfo();
19      v_UI_2.showInfo();
20  </script>
```

关于【代码 15-3】的分析如下：

【代码 15-3】与【代码 15-2】的主要区别就是，第 02～13 行代码通过一个函数 createUserInfo()的方式将第 03～11 行代码定义“类”的过程封装了进去。同时，第 12 行代码通过函数返回值的方式返回了对象实例 userInfo。

第 15 和第 18 行代码分别通过调用函数 createUserInfo()的方式，定义了两个 userInfo 类的对象实例。

第 16 和第 19 行代码通过调用 userInfo 类的 showInfo()方法，在浏览器控制台中输出对象实例的内容。

运行页面，控制台输出的调试信息如图 15.3 所示。第 15 和第 18 行代码定义的两个对象实例输出了相同的结果，这样就解决了“类”重复使用的问题。

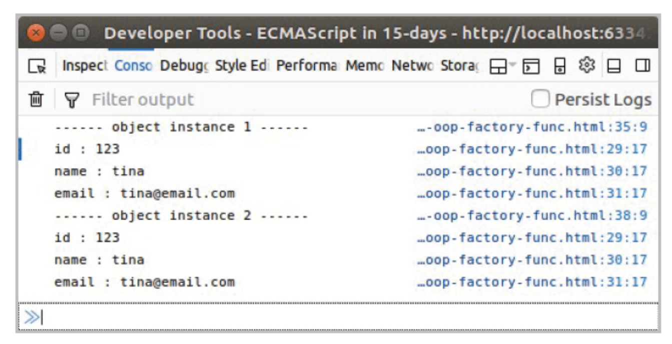


图 15.3　封装的工厂模式创建类和对象

不过，【代码 15-3】创建“类”的方式还是有一些小问题，“类”的属性值都是固定的，难道每次使用不同的属性值时都要重新定义“类”吗？其实解决方式很简单，只要为封装函数带上参数就可以了。

15.3.3　带参数的工厂模式创建类与对象

为封装函数带上参数与我们通常定义带参数的 ECMAScript 函数是一样的，这样就可以在实例化对象时定义不同的属性值了。

下面是一个定义封装形式的、并有函数参数的 ECMAScript 类（对象）的代码示例（详见源代码 ch15 目录中 ch15-es-oop-factory-func-param.html 文件）。

【代码 15-4】

```
01  <script type="text/javascript">
02      function createUserInfo(id, name, email) {
03          var userInfo = new Object;
04          userInfo.id = id;
05          userInfo.name = name;
06          userInfo.email = email;
07          userInfo.showInfo = function () {
08              console.log("id : " + this.id);
09              console.log("name : " + this.name);
10              console.log("email : " + this.email);
11          };
12          return userInfo;
13      }
14      console.log("------ object instance 1 ------");
15      var v_UI_1 = createUserInfo("123", "tina", "tina@email.com");
16      v_UI_1.showInfo();
17      console.log("------ object instance 2 ------");
18      var v_UI_2 = createUserInfo("520", "xixi", "xixi@email.com");
19      v_UI_2.showInfo();
20  </script>
```

关于【代码 15-4】的分析如下：

【代码 15-4】与【代码 15-3】的主要区别就是，第 02～13 行代码在定义函数 createUserInfo()的方式中增加了函数参数的定义。

第 15 和第 18 行代码在调用函数 createUserInfo()的过程中，通过定义参数值进行了初始化操作。

运行页面，控制台输出的调试信息如图 15.4 所示。第 15 行和第 18 行代码定义的两个对象实例输出了不同的结果，这样就解决了“类”初始化不同属性值的问题。

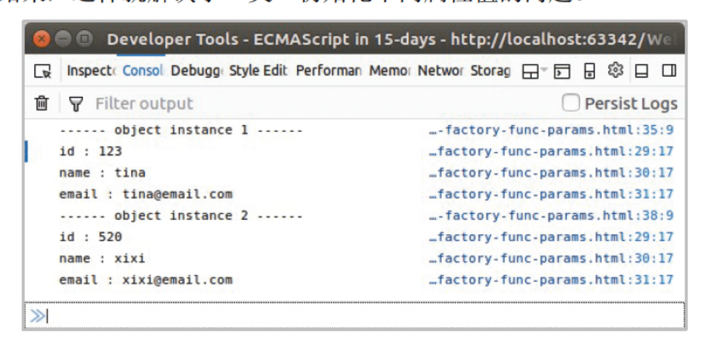


图 15.4　带参数的工厂模式创建类和对象

15.3.4 工厂模式的最大局限

经过前面的介绍，似乎通过工厂模式（Factory）创建 ECMAScript 类和对象已经可以解决问题了，其实是经不住严格推敲的。再回到【代码 15-4】中，每当创建一个 userInfo 对象实例，第 07～11 行代码定义的方法就会被重新创建一次，即每一个 userInfo 对象实例都会有属于自己的 showInfo 方法。这也正是纯工厂模式的最大局限（可以参阅一些关于设计模式的教科书）。

那么怎么解决这个问题呢？最直接的方法就是将 showInfo 方法的定义放在工厂函数 createUserInfo()的外面，然后通过定义“类”属性的方式指向该函数（类似于函数指针）。

下面是一个按照上述解决办法来定义 ECMAScript 类（对象）的代码示例（详见源代码 ch15 目录中的 ch15-es-oop-factory-func-re.html 文件）。

【代码 15-5】

```
01  <script type="text/javascript">
02      function showInfo() {
03          console.log("id : " + this.id);
04          console.log("name : " + this.name);
05          console.log("email : " + this.email);
06      }
07      function createUserInfo(id, name, email) {
08          var userInfo = new Object;
09          userInfo.id = id;
10          userInfo.name = name;
11          userInfo.email = email;
12          userInfo.showInfo = showInfo;
13          return userInfo;
14      }
15      console.log("------ object instance 1 ------");
16      var v_UI_1 = createUserInfo("123", "tina", "tina@email.com");
17      v_UI_1.showInfo();
18      console.log("------ object instance 2 ------");
19      var v_UI_2 = createUserInfo("520", "xixi", "xixi@email.com");
20      v_UI_2.showInfo();
21  </script>
```

关于【代码 15-5】的分析如下：

【代码 15-5】与【代码 15-4】的主要区别就是，第 12 行代码定义的对象实例 userInfo 的 showInfo 属性，该属性所指向的属性值 showInfo 对应第 01～06 行代码定义函数 showInfo()。

通过上面这种方式就避免了“类”的方法被对象实例重复创建的过程，这样每定义一个 userInfo 对象实例，showInfo 属性值就相当于一个函数指针，均指向第 01～06 行代码定义函数 showInfo()。

运行页面，控制台输出的调试信息如图 15.5 所示。从浏览器控制台的输出结果来看，【代码 15-5】与【代码 15-4】的效果是完全一致的。

图 15.5　改进工厂模式的局限

15.3.5　构造函数方式创建类与对象

工厂模式是程序语言设计模式的一大进步，但正如前面的代码实例所显示的结果一样，该方式也有一定的局限性。于是，设计人员又提出通过构造函数方式来创建 ECMAScript 类与对象。

下面是一个通过构造函数方式定义 ECMAScript 类（对象）的代码示例（详见源代码 ch15 目录中的 ch15-es-oop-constructor.html 文件）。

【代码 15-6】

```
01  <script type="text/javascript">
02      function UserInfo(id, name, email) {
03          this.id = id;
04          this.name = name;
05          this.email = email;
06          this.showInfo = function () {
07              console.log("id : " + this.id);
08              console.log("name : " + this.name);
09              console.log("email : " + this.email);
10          };
11      }
12      console.log("------ new object instance 1 ------");
13      var v_UI_1 = new UserInfo("123", "tina", "tina@email.com");
14      v_UI_1.showInfo();
15      console.log("------ new object instance 2 ------");
16      var v_UI_2 = new UserInfo("520", "xixi", "xixi@email.com");
17      v_UI_2.showInfo();
18  </script>
```

关于【代码 15-6】的分析如下：

【代码 15-6】与【代码 15-4】的主要区别就是，第 02～11 行代码通过一个构造函数 UserInfo() 的方式定义了一个类 UserInfo。注意，一般需要将类名称首字母大写，以区别函数方法。

从第 02～11 行代码中可以看到，构造函数方式下既没有创建（new）对象也没有定义返回值。同时，构造函数内的属性都是通过 this 关键字来引用的。

在第 13 行和第 16 行代码中两个 UserInfo 类的对象实例都是通过 new 关键字来创建的，这也是构造函数方式与工厂模式的主要区别之一。

运行页面，控制台输出的调试信息如图 15.6 所示。【代码 15-6】和【代码 15-4】输出的结果

是完全相同的，而且【代码 15-6】的代码结构与形式更像传统意义上的面向对象语言（如 C++、Java 等）。

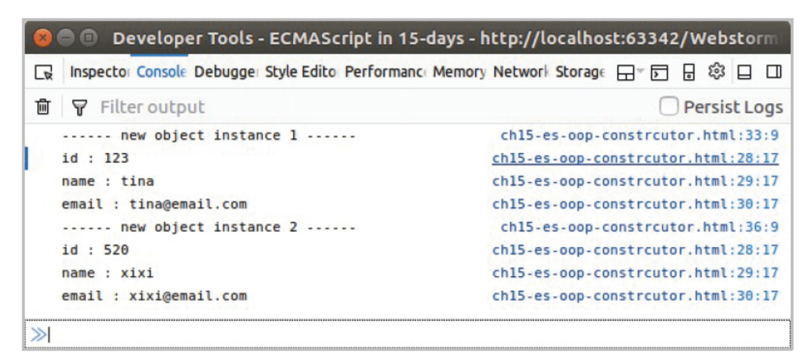


图 15.6　构造函数方式创建类和对象

不过，【代码 15-6】创建类的方式与前面几个代码实例类似，也会出现类的方法被重复定义的问题。那么有没有更简单、更合理的方式呢？答案是肯定的，我们接着往下看。

15.3.6　原型方式创建类与对象

ECMAScript 脚本语言最特别的就是提供了一个原型（prototype）属性。具体来说，原型（prototype）属性是由 Object 对象提供的，且可以被其他对象所继承。

前面也多次提到了 ECMAScript 脚本语言中“一切皆为对象”的概念，因此任何一个对象也自然会有原型（prototype）属性。ECMAScript 脚本语言通过原型（prototype）属性可以添加属性和方法，即借助原型（prototype）方式就可以创建类与对象。

下面是一个通过原型（prototype）方式定义 ECMAScript 类（对象）的基本代码示例（详见源代码 ch15 目录中的 ch15-js-oop-prototype.html 文件）。

【代码 15-7】

```
01  <script type="text/javascript">
02      function UserInfo() {
03      }
04      UserInfo.prototype.id = "123";
05      UserInfo.prototype.name = "tina";
06      UserInfo.prototype.email = "tina@email.com";
07      UserInfo.prototype.showInfo = function () {
08          console.log("id : " + this.id);
09          console.log("name : " + this.name);
10          console.log("email : " + this.email);
11      };
12      console.log("------ new object instance 1 ------");
13      var v_UI_1 = new UserInfo();
14      v_UI_1.showInfo();
15      console.log("------ new object instance 2 ------");
16      var v_UI_2 = new UserInfo();
17      v_UI_2.showInfo();
18  </script>
```

关于【代码 15-7】的分析如下：

原型方式的特点就是如第 02 和第 03 行代码先定义一个空的构造函数 UserInfo()，然后如第 04～11 行代码通过 prototype 属性为这个构造函数 UserInfo()定义属性和方法，从而完成 ECMAScript 类（对象）的创建过程。

运行页面，控制台输出的调试信息如图 15.7 所示。从浏览器控制台的输出结果来看，第 13 和第 16 行代码定义的对象实例输出了相同的内容，主要是因为构造函数 UserInfo()没有定义参数造成的。

图 15.7　原型方式创建类和对象

那么能不能解决带参数这个问题呢？根据原型方式的特点，属性和方法均是通过原型（prototype）属性来定义的，单单依靠原型方式确实有些困难。不过，将原型方式与构造函数方式结合起来，还是能够解决问题的，我们接着往下看。

15.3.7　结合构造函数方式与原型方式创建类和对象

前面已经详细介绍了通过构造函数方式创建 ECMAScript 类与对象，以及通过对象与原型方式创建 ECMAScript 类与对象的过程。本小节结合上述两种方式来实现全新的创建 ECMAScript 类与对象的方式。

下面是一个通过结合构造函数方式与原型（prototype）方式定义 ECMAScript 类（对象）的代码示例（详见源代码 ch15 目录中的 ch15-es-oop-constructor-prototype.html 文件）。

【代码 15-8】

```
01  <script type="text/javascript">
02      function UserInfo(id, name, email) {
03          this.id = id;
04          this.name = name;
05          this.email = email;
06      }
07      UserInfo.prototype.showInfo = function () {
08          console.log("id : " + this.id);
09          console.log("name : " + this.name);
10          console.log("email : " + this.email);
11      };
12      console.log("------ new object instance 1 ------");
```

```
13        var v_UI_1 = new UserInfo("123", "tina", "tina@email.com");
14        v_UI_1.showInfo();
15        console.log("------ new object instance 2 ------");
16        var v_UI_2 = new UserInfo("520", "xixi", "xixi@email.com");
17        v_UI_2.showInfo();
18    </script>
```

关于【代码 15-8】的分析如下：

第 02～06 行代码通过带参数的构造函数方式定义了一个 ECMAScript 类 UserInfo()，与【代码 8-6】类似。

第 07～11 行代码通过原型（prototype）方式为上面的构造函数 UserInfo()定义方法 showInfo，从而完成 ECMAScript 类（对象）的创建过程。

运行页面，控制台输出的调试信息如图 15.8 所示。从浏览器控制台的输出结果来看，通过结合构造函数方式与原型方式创建 ECMAScript 类（对象）是目前非常理想的一种方式。

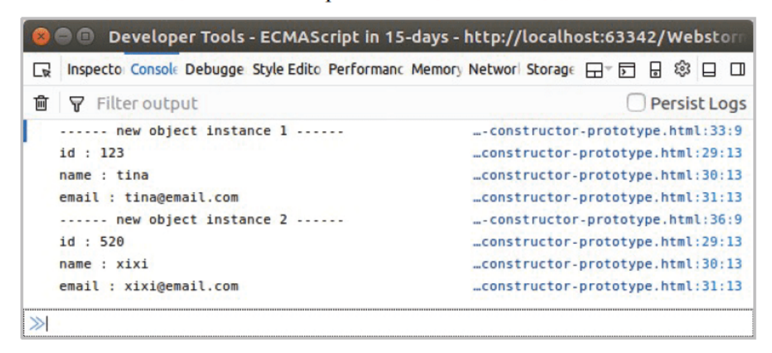


图 15.8　结合构造函数方式与原型方式创建类和对象

15.4　原型 Prototype 应用

本节将介绍原型（Prototype）属性的几种应用方法，包括为 ECMAScript 对象定义新方法、重定义已有方法和实现类继承等。

15.4.1　定义新方法

在 ECMAScript 语法规范中，原型（Prototype）属性是一个很有用的工具。在实际开发中，可以通过原型（Prototype）属性为 ECMAScript 规范中常规的对象定义新的方法，具有很强的扩展功能。

下面是通过原型（prototype）属性定义一个 String 对象新方法的代码示例（详见源代码 ch15 目录中的 ch15-es-oop-prototype-new-method.html 文件）。

【代码 15-9】

```
01    <script type="text/javascript">
02        var v_str = "Prototype";
```

```
03          console.log("------ toString() ------");
04          console.log(v_str.toString());
05          String.prototype.toReverseString = function () {
06              return this.split("").reverse().join("");
07          };
08          console.log("------ toReverseString() ------");
09          console.log(v_str.toReverseString());
10      </script>
```

关于【代码 15-9】的分析如下：

ECMAScript 脚本语言中针对 String 对象并没有直接进行倒序的方法，这段代码就是通过原型属性为 String 对象添加倒序处理方法 toReverseString 的过程。

第 05～07 行代码通过原型属性为 String 对象定义了一个新方法 toReverseString，从函数命名规范来看正是与 toString()方法相对应的。

第 06 行代码先通过调用 split()方法将 String 对象转换为单字符的数组，然后通过 reverse("")方法对数组进行倒序操作（注意 reverse()方法是 Array 对象的方法），最后通过 join("")方法将字符数组连接成字符串格式并返回。

运行页面，控制台输出的调试信息如图 15.9 所示。如图 15.9 中箭头所指，从浏览器控制台的输出结果来看，通过调用 toReverseString()方法已成功将字符串进行倒序输出。

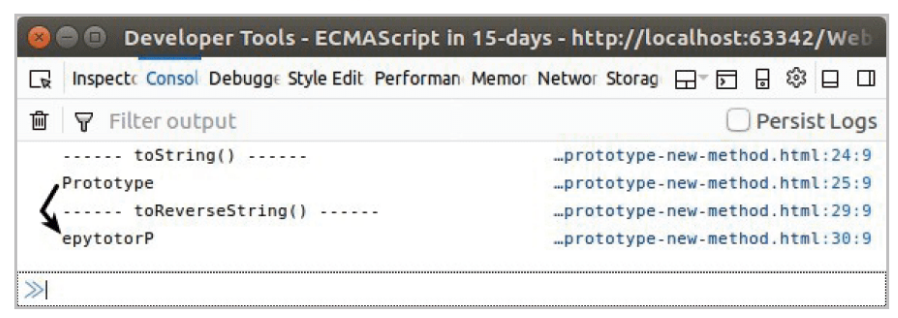


图 15.9　通过原型（Prototype）属性定义对象新方法

15.4.2　重定义已有方法

通过原型（Prototype）属性除了可以为 ECMAScript 语法规范中常规的对象定义新的方法外，还可以为常规的对象重定义已有的方法。

下面是通过原型属性为 String 对象重定义一个已有方法的代码示例（详见源代码 ch15 目录中的 ch15-es-oop-prototype-revise-method.html 文件）。

【代码 15-10】

```
01      <script type="text/javascript">
02          var v_str = "Prototype";
03          console.log("------ print string ------");
04          console.log(v_str);
05          console.log("------ toString() ------");
06          console.log(v_str.toString());
07          String.prototype.toString = function () {
08              return "revise toString() : " + this;
```

```
09          };
10          console.log("------ print string after revise toString() ------");
11          console.log(v_str);
12          console.log("------ revise toString() ------");
13          console.log(v_str.toString());
14      </script>
```

关于【代码 15-10】的分析如下：

对于一个 String 对象来说，使用或不使用 toString()方法进行输出的结果是一样的，这段代码就是通过原型属性为 String 对象的 toString()方法进行了重定义，并测试了重定义后 toString()方法的效果。

第 07～09 行代码通过原型属性为 String 对象重定义了 toString()方法。第 08 行代码在返回原始字符串（关键字 this 引用的）前插入了前缀字符串“revise toString() : ”。

运行页面，控制台输出的调试信息如图 15.10 所示。在重定义 toString()方法之前，对 String 对象使用或不使用 toString()方法进行输出的结果是一样的。而在重定义 toString()方法之后，对 String 对象是否显式地调用 toString()方法进行输出的结果是不同的。从第 13 行代码输出的结果来看，通过原型属性重定义 toString()方法之后，输出的结果已经发生了改变，这也正是原型属性功能强大的又一体现。

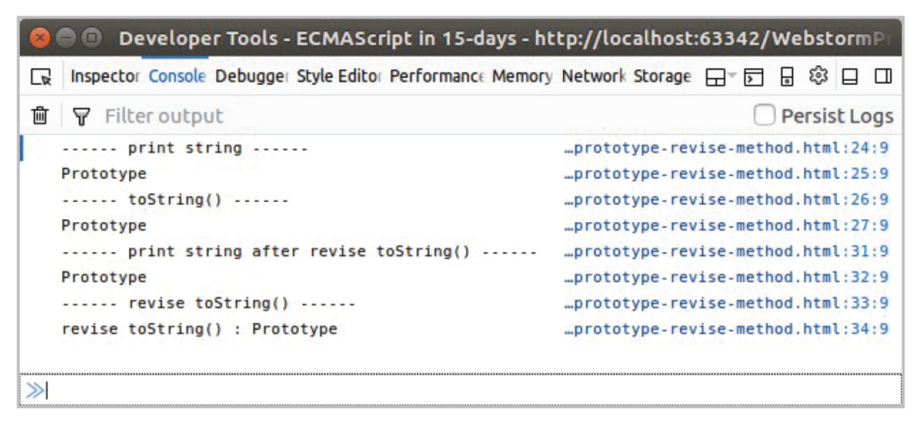


图 15.10　通过原型（Prototype）属性重定义对象已有的方法

15.4.3　实现继承机制

原型（Prototype）属性还有一个很重要的功能，就是实现 ECMAScript 类（对象）的继承机制。

下面是一个通过原型属性实现继承机制的代码示例（详见源代码 ch15 目录中的 ch15-es-oop-prototype-inherit.html 文件）。

【代码 15-11】

```
01  <script type="text/javascript">
02      /**
03       * 定义基类——ClassBase
04       * @param id
05       * @param name
06       * @param email
```

```
07        * @constructor
08        */
09       function ClassBase(id, name, email) {
10          this.id = id;
11          this.name = name;
12          this.email = email;
13       }
14       ClassBase.prototype.showInfo = function () {
15          console.log("id : " + this.id);
16          console.log("name : " + this.name);
17          console.log("email : " + this.email);
18       };
19       console.log("------ new ClassBase ------");
20       var v_cb = new ClassBase("123", "tina", "tina@email.com");
21       v_cb.showInfo();
22       /**
23        * 定义子类——ClassInheritA
24        * @param id
25        * @param name
26        * @param email
27        * @constructor
28        */
29       function ClassInheritA(id, name, email) {
30          ClassBase.call(this, id, name, email);
31       }
32       ClassInheritA.prototype = new ClassBase();
33       console.log("------ new ClassInheritA ------");
34       var v_ci_A = new ClassInheritA("520", "xixi", "xixi@email.com");
35       v_ci_A.showInfo();
36       /**
37        * 定义子类——ClassInheritB
38        * @param id
39        * @param name
40        * @param email
41        * @param title
42        * @constructor
43        */
44       function ClassInheritB(id, name, email, title) {
45          ClassBase.call(this, id, name, email);
46          this.title = title;
47       }
48       ClassInheritB.prototype = new ClassBase();
49       ClassInheritB.prototype.showInfo = function () {
50          console.log("id : " + this.id);
51          console.log("name : " + this.name);
52          console.log("email : " + this.email);
53          console.log("title : " + this.title);
54       };
55       console.log("------ new ClassInheritB ------");
56       var v_ci_B = new ClassInheritB("985", "dudu", "dudu@email.com", "Boss");
57       v_ci_B.showInfo();
58    </script>
```

关于【代码 15-11】的分析如下：

这段代码主要就是通过原型属性实现 ECMAScript 类（对象）的继承机制。

第 09～13 行和第 14～18 行代码通过构造函数方式和原型方式创建了一个 ECMAScript 类 ClassBase，可以称其为“基类或父类”，这段代码与【代码 8-8】的定义过程基本一致。

第 29～31 行代码通过构造函数方式创建了第一个 ECMAScript 类 ClassInheritA，可以称其为“继承类或子类”，该类是创建的第一个子类。

第 30 行代码调用 Function 对象的 call()方法，将基类 ClassBase 定义的几个属性继承到子类 ClassInheritA 中。

第 32 行代码通过 new 关键字，将子类 ClassInheritA 的原型属性定义为基类 ClassBase 的对象实例，实现了子类 ClassInheritA 对基类 ClassBase 的继承。

另外，关于继承机制的原理，读者可以参考通过原型属性和原型链的内容来了解。

第 44～47 行代码通过构造函数方式创建了第二个 ECMAScript 类 ClassInheritB，与第一个 ECMAScript 类 ClassInheritA 不同的地方是增加了一个 title 属性。注意，该属性是仅属于子类 ClassInheritB 的。

第 45 行代码与第 30 行代码功能一样，将基类 ClassBase 定义的几个属性继承到子类 ClassInheritB 中。

第 46 行代码对 title 属性进行初始化操作。

第 48 行代码与第 32 行代码功能一样，通过 new 关键字将子类 ClassInheritB 的 prototype 属性定义为基类 ClassBase 的对象实例。

第 49～54 行代码对子类 ClassInheritB 的 showInfo 方法进行了重定义，因为子类 ClassInheritB 新增加了一个 title 属性，所以自然基类 ClassBase 的 showInfo 方法也就不适用于子类 ClassInheritB 了。

运行页面，控制台输出的调试信息如图 15.11 所示。从浏览器控制台的输出结果来看，子类 ClassInheritA 继承了基类 ClassBase 的全部属性和方法；子类 ClassInheritB 在继承了基类 ClassBase 的全部属性和方法外，还成功添加了属于自己的属性。

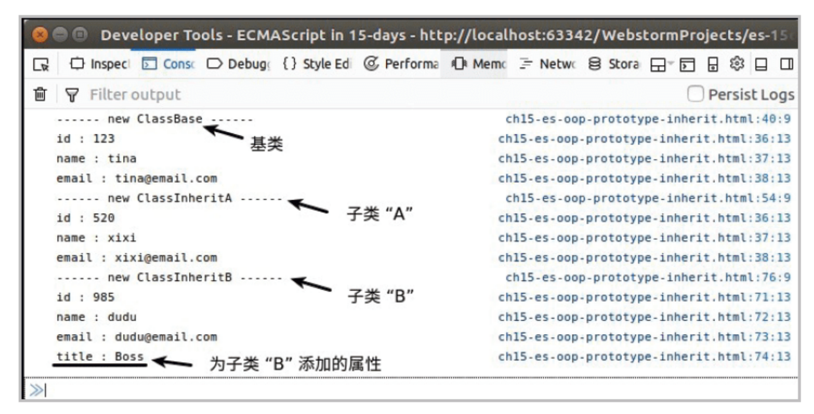


图 15.11　通过原型（Prototype）属性实现继承机制

对于学习过 C++或 Java 语言面向对象编程的读者来说，【代码 15-11】中所实现的 ECMAScript 类继承已经很接近 C++或 Java 语言了，但还是有一点被设计人员所“吐槽”的地方，为什么没有 class 关键字直接定义类呢？

也许是为了响应广大设计人员的诉求吧，当前的 ECMAScript 6 版本规范中，就真的增加 class

关键字定义“类”的功能。那么这个“class”是不是与 C++或 Java 语言中的“class”一样呢，请读者继续往下阅读。

15.5　ECMAScript 6 面向对象新特性

本节将介绍 ECMAScript 6 语法规范中新增的面向对象特性——class 关键字。

15.5.1　通过“class”定义类

在 ECMAScript 6 语法规范中，可以通过 class 关键字创建并定义“类”。这个 ECMAScript 6 的新特性对于广大学习过 C++或 Java 语言的读者来说，是一个令人振奋的消息。

虽然 ECMAScript 6 提供的 class 关键字能够模仿传统定义“类”的方式，但其内部原理仍旧是通过原型（prototype）方式来实现的。

下面是通过 class 关键字定义一个类的代码示例（详见源代码 ch15 目录中的 ch15-es-oop-class.html 文件）。

【代码 15-12】

```
01  <script type="text/javascript">
02      class UserInfo {
03         constructor(id, name, email) {
04            this.id = id;
05            this.name = name;
06            this.email = email;
07         }
08         showInfo() {
09            console.log("id : " + this.id);
10            console.log("name : " + this.name);
11            console.log("email : " + this.email);
12         }
13      }
14      console.log("------ new class instance 1 ------");
15      var v_UI_1 = new UserInfo("123", "tina", "tina@email.com");
16      v_UI_1.showInfo();
17      console.log("------ new class instance 2 ------");
18      var v_UI_2 = new UserInfo("520", "xixi", "xixi@email.com");
19      v_UI_2.showInfo();
20  </script>
```

关于【代码 15-12】的分析如下：

第 02～13 行代码通过 class 关键字定义了一个类 UserInfo，看上去与 C++和 Java 语言定义“类”的方式是一样的。

第 03～07 行代码通过关键字 constructor 定义了一个构造方法，方法内定义了一组参数，用于表示类 UserInfo 的属性。其中，第 04～06 行代码通过 this 关键字为属性进行了初始化操作。

第 08～12 行代码为类 UserInfo 定义了方法 showInfo。注意，此处定义方法时无须使用 function

关键字。

运行页面，控制台输出的调试信息如图 15.12 所示。从浏览器控制台的输出结果来看，【代码 15-12】与【代码 15-8】所实现的功能和效果是一致的。

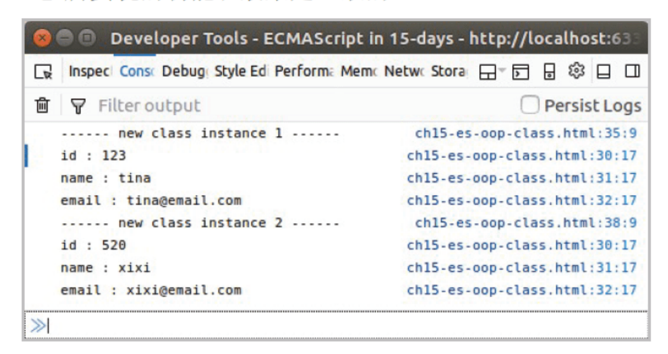


图 15.12 通过 class 创建类

15.5.2 通过“extends”继承类

既然 ECMAScript 6 版本语法规范中已可以通过 class 关键字来创建并定义“类”了，自然也实现了“类”的继承。与其他面向对象高级语言一样，ECMAScript 6 语法规范中同样使用了 extends 关键字来定义类继承。

下面是通过 extends 关键字定义一个类继承的代码示例（详见源代码 ch15 目录中的 ch15-es-oop-class-extends.html 文件）。

【代码 15-13】

```
01  <script type="text/javascript">
02      /**
03       * class --- UserInfo
04       */
05      class UserInfo {
06          constructor(id, name, email) {
07              this.id = id;
08              this.name = name;
09              this.email = email;
10          }
11          showInfo() {
12              console.log("id : " + this.id);
13              console.log("name : " + this.name);
14              console.log("email : " + this.email);
15          }
16      }
17      console.log("------ new base class instance ------");
18      var v_user = new UserInfo("123", "tina", "tina@email.com");
19      v_user.showInfo();
20      /**
21       * class --- ManagerInfo extend UserInfo
22       */
23      class ManagerInfo extends UserInfo {
24          constructor(id, name, email, title) {
```

```
25              super(id, name, email);
26              this.title = title;
27          }
28          showInfo() {
29              super.showInfo();
30              console.log("title : " + this.title);
31          }
32      }
33      console.log("------ new sub class instance ------");
34      var v_manager = new ManagerInfo("xixi", "xixi", "xixi@email.com", "princess");
35      v_manager.showInfo();
36  </script>
```

关于【代码 15-13】的分析如下：

【代码 15-13】是在【代码 15-12】基础上修改而成的，主要就是类 ManagerInfo 对类 UserInfo 继承的实现。

第 05～16 行代码是定义父类 UserInfo 的过程，与【代码 15-12】中的定义过程完全一致。

第 23～32 行代码是定义子类 ManagerInfo 的过程，重点就是使用关键字 extends 来继承父类 UserInfo。

第 24～27 行代码通过关键字 constructor 定义了一个构造方法，方法内定义了一组参数，用于表示子类 ManagerInfo 的属性。注意，相比父类 UserInfo 的属性，这里增加了一个 title 属性。

第 25 行代码通过 super 方法默认调用父类 UserInfo 的构造方法，这行代码是必须写的，否则就会出现错误。

第 26 行代码通过 this 关键字为 title 属性进行了初始化操作。

第 28～31 行代码为子类 ManagerInfo 定义了方法 showInfo，由于父类也有一个同名的方法，这里的方法 showInfo 相当于一个重载方法。因此，第 29 行代码先通过 super 方法默认调用父类 UserInfo 的方法 showInfo，然后在第 30 行代码重写对 title 属性的操作。

运行页面，控制台输出的调试信息如图 15.13 所示。从浏览器控制台的输出结果来看，【代码 15-13】实现了“类”的继承操作。

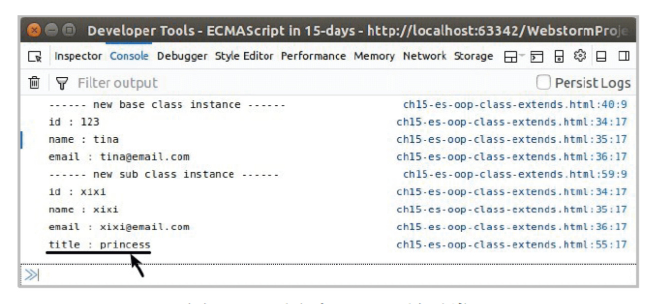


图 15.13　通过 extends 继承类

15.5.3　类的 setter 和 getter 方法

严格来说，ECMAScript 6 语法规范中关于“类”的存值 setter 方法和取值 getter 方法在 ECMAScript 5 版本规范中就已经存在了。这里介绍存值 setter 方法和取值 getter 方法，是为了更好

地与 ECMAScript 6 新增的类 class 特性结合使用。

下面是一个在类 class 中使用存值 setter 方法和取值 getter 方法的代码示例（详见源代码 ch15 目录中的 ch15-es-oop-class-set-get.html 文件）。

【代码 15-14】

```
01  <script type="text/javascript">
02      class UserInfo {
03          constructor() {
04          }
05          get id() {
06              return this._id_;
07          }
08          set id(value) {
09              this._id_ = value;
10          }
11      }
12      console.log("------ new class instance ------");
13      var v_ui = new UserInfo();
14      console.log(v_ui.id);
15      v_ui.id = "id-001";
16      console.log("id : " + v_ui.id);
17      console.log("_id_ : " + v_ui._id_);
18  </script>
```

关于【代码 15-14】的分析如下：

第 02～13 行代码通过关键字 class 定义了一个类 UserInfo。

第 03～04 行代码通过关键字 constructor 定义了一个空的默认构造方法。

第 05～07 行和第 08～10 行代码分别通过 getter 方法和 setter 方法定义了一个属性 id 的存储器。

运行页面，控制台输出的调试信息如图 15.14 所示。从浏览器控制台的输出结果来看，在未通过 setter 方法为属性 id 存储数据前，通过 getter 方法获取的属性 id 值为 undefined。而在通过 setter 方法为属性 id 存储数据“id-001”后，通过 getter 方法成功获取了属性 id 值“id-001”。

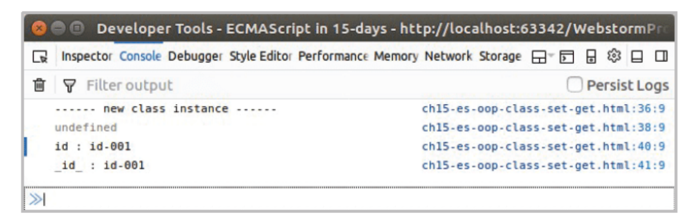


图 15.14　类的 setter 方法和 getter 方法

15.6　本章小结

本章主要介绍了 ECMAScript 6 语法规范中关于面向对象编程的知识，包括面向对象基础、ECMAScript 对象应用、ECMAScript 对象类型，以及一些 ECMAScript 6 类（class）新特性的知识，并通过一些具体实例进行了讲解。

第 16 章

ECMAScript 7 & 8 版本新特性

最后一章将介绍一下 ECMAScript 7 & 8 版本所带来的新特性。虽然，ECMAScript 7 & 8 版本更新的内容不多，但还是增加了几个非常重要的新功能，这几个新特性有助于设计人员提高编程效率。

16.1　ECMAScript 7 & 8 版本的新特性

众所周知，ECMAScript 6（或 ECMAScript 2015）版本是目前 ECMAScript 语法规范最重要的、也是内容最丰富的一次版本更新。因此，通常意义上所讲的 ECMAScript 最新版语法规范也是指 ECMAScript 6 版本所更新的内容。

但是，我们可能无意识地忽略了 ECMA 国际（TC39，JavaScript 技术委员会）的进取心。其实，自从 ECMAScript 6 正式版本于 2015 年正式发布以来，TC39 制定了每年进行一次版本更新的计划。

因此，TC39 在 ECMAScript 6 版本之后，还推出了 ECMAScript 7（或 ECMAScript 2016）和 ECMAScript 8（或 ECMAScript 2017）两个重要版本。同时，为了保证 ECMAScript 语法规范每年更新一次的计划得以实现，在 ECMAScript 6 版本之后，版本编号将会按照年份来命名。也就是说，新版 ECMAScript 语法规范的正式名称应该是 ECMAScript 2016 和 ECMAScript 2017 版本。

16.2　ECMAScript 7（2016）版本的新特性

ECMAScript 7（2016）版本语法规范中增加的新特性只有两项，分别是数组对象的 includes() 方法和指数操作符。

16.2.1 Array.prototype.includes()方法

Array.prototype.includes()方法是 ECMAScript 7 版本语法规范中新增的特性，该方法的基本语法格式如下：

```
Array.prototype.includes(value : any)
```

该方法用于检查 value 值是否是数组 Array 的数组项，若是，则返回 true，否则返回 false。

读者可能会注意到，Array 对象有一个 indexOf()方法与 includes()方法功能上很类似，其实两者还是有区别的。下面看一个对比使用 indexOf()方法与 includes()方法的代码示例（详见源代码 appendix 目录中的 appendix-es-7-array-includes.html 文件）。

【代码 16-1】

```
01 <script type="text/javascript">
02 'use strict';
03      /*
04       * ECMAScript 7（2016）新特性 - includes()方法
05       */
06      console.log("---------- Search Array ----------");
07      let arr = ['Hello', 'ECMAScript', '!'];
08      if(arr.indexOf('ECMAScript') !== -1)
09         console.log("Array.indexOf('ECMAScript') return true.");
10      else
11         console.log("Array.indexOf('ECMAScript') return false.");
12      if(arr.includes('ECMAScript'))
13         console.log("Array.includes('ECMAScript') return true.");
14      else
15         console.log("Array.includes('ECMAScript') return false.");
16      console.log("---------- Search NaN ----------");
17      let arrNaN = [NaN];
18      if(arrNaN.indexOf(NaN) !== -1)
19         console.log("Array.indexOf(NaN) return true.");
20      else
21         console.log("Array.indexOf(NaN) return false.");
22      if(arrNaN.includes(NaN))
23         console.log("Array.includes(NaN) return true.");
24      else
25         console.log("Array.includes(NaN) return false.");
26 </script>
```

关于【代码 16-1】的分析如下：

第 07 行代码定义了一个字符串数组 arr 变量。

第 08～11 行代码通过 Array 对象的 indexOf()方法查询数组变量 arr 中是否包含指定的数组项'ECMAScript'。注意，在使用 indexOf()方法时，要通过与数值-1 进行严格比较（!==）来判断查询结果。

第 12～15 行代码通过 Array 对象新增的 includes()方法查询数组变量 arr 中是否包含指定的数组项'ECMAScript'。注意，在使用 includes()方法时，是不需要与数值-1 进行严格比较（!==）就可以得到查询结果的。

第 17 行代码定义了一个特殊值数组 arrNaN 变量，该数组定义了一个特殊值的数组项 NaN。

第 18～21 行代码与第 08～11 行代码类似，通过 Array 对象的 indexOf()方法查询数组变量 arrNaN 中是否包含指定的特殊值数组项 NaN。

第 22～25 行代码与第 12～15 行代码类似，通过 Array 对象新增的 includes()方法查询数组变量 arrNaN 中是否包含指定的特殊值数组项 NaN。

下面运行测试一下【代码 16-1】所定义的 HTML 页面，并使用调试器查看控制台输出的调试信息，页面效果如图 16.1 所示。

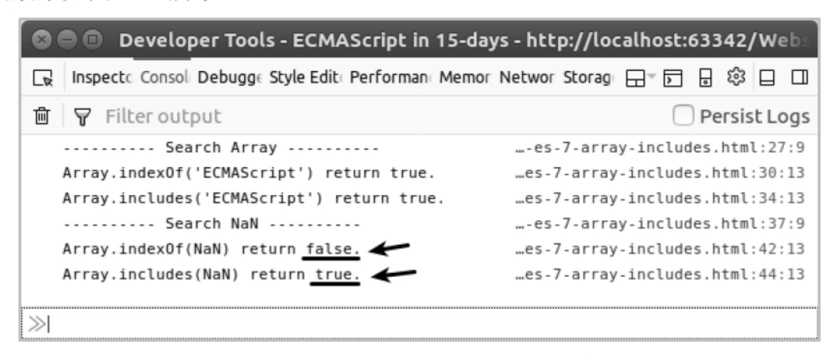


图 16.1　Array.prototype.includes()方法

如图 16.1 所示，在使用 indexOf()方法与 includes()方法查询常规数组项（如字符串类型）时，功能是基本一致的。唯一的区别就是，includes()方法在使用上要比 indexOf()方法简单一些。那么，indexOf()方法与 includes()方法之间的主要区别是什么呢？

如图 16.1 中箭头所指，在使用 indexOf()方法与 includes()方法查询特殊值数组项（如 NaN）时，includes()方法就发挥不出作用了（查询结果返回 false），而 includes()方法是完全可以胜任的（查询结果返回 true）。

16.2.2　指数操作符

在 ECMAScript 6 版本语法规范中，指数运算一般是通过全局 Math 对象的 pow()方法来实现的。而在 ECMAScript 7 版本语法规范中，新定义了一个指数运算符（**，用连续两个*号表示），这样就极大地优化了编写指数运算的代码量。

下面看一个使用指数运算符（**）的代码示例（详见源代码 appendix 目录中的 appendix-es-7-indices.html 文件）。

【代码 16-2】

```
01 <script type="text/javascript">
02 'use strict';
03       /*
04        * ECMAScript 7（2016）新特性 - 指数运算
05        */
06       console.log("---------- 指数运算 ----------");
07       var iIndices = 2**10;
08       console.log("2 ^ 10 = " + iIndices);
09 </script>
```

关于【代码 16-2】的分析如下：

第 07 行代码通过指数运算符**计算了数值 2 的 10 次幂。

下面运行测试一下【代码 16-2】所定义的 HTML 页面，并使用调试器查看控制台输出的调试信息，页面效果如图 16.2 所示。

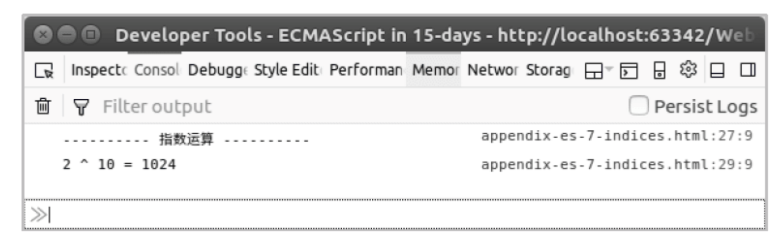


图 16.2　指数运算符（**）

16.3　ECMAScript 8（2017）版本的新特性

ECMAScript 8（2017）版本语法规范中增加的新特性相对要多一些，下面重点列举几个实用的特性。

16.3.1　字符串填充（String Padding）

ECMAScript 8（2017）版本语法规范中新增了一组内置的、用于字符串填充的方法，分别为 padStart()和 padEnd()。这一组方法能够通过填充字符串的首部或尾部来保证字符串达到固定的长度，并完成字符串的插入功能。

关于 padStart()方法的基本语法格式如下：

```
String.padStart(targetLength [, padString])
```

参数说明：

- targetLength：该参数是必需的，用于表示字符串要设定的目标长度，即最终生成的字符串长度。
- padString：该参数是可选的，用于指定所需填充的字符串。

关于 padEnd()方法的基本语法格式如下：

```
String.padEnd(targetLength [, padString])
```

参数说明：

- targetLength：该参数是必需的，用于表示字符串要设定的目标长度，即最终生成的字符串长度。
- padString：该参数是可选的，用于指定所需填充的字符串。

下面看一个使用 padStart()方法与 padEnd()方法完成字符串填充的代码示例（详见源代码

appendix 目录中的 appendix-es-7-string-padding.html 文件）。

【代码 16-3】

```
01 <script type="text/javascript">
02 'use strict';
03      /*
04       * ECMAScript 8 (2017) 新特性 - 字符串填充 (String Padding)
05       */
06      console.log("---------- 字符串填充 (String Padding) ----------");
07      var str = "";
08      str = str.padStart(4, "ECMA");
09      console.log("str.padStart(4, \"ECMA\") = " + str);
10      str = str.padStart(5);
11      console.log("str.padStart(5) = " + str);
12      str = str.padStart(10, "Hello");
13      console.log("str.padStart(10, \"Hello\") = " + str);
14      str = str.padEnd(16, "Script");
15      console.log("str.padEnd(16, \"Script\") = " + str);
16      str = str.padEnd(17);
17      console.log("str.padEnd(17) = " + str);
18      str = str.padEnd(18, "!");
19      console.log("str.padEnd(18, \"!\") = " + str);
20 </script>
```

关于【代码 16-3】的分析如下：

第 07 行代码定义了一个空字符串变量 str。

第 08 行代码通过使用 padStart(4, "ECMA")方法，先设定字符串目标长度为 4，然后在首部插入字符串"ECMA"。

第 10 行代码通过使用 padStart(5)方法再次从字符串首部设定字符串目标长度为 5。

第 12 行代码通过使用 padStart(10, "Hello")方法再次设定字符串目标长度为 10，然后在首部插入字符串"Hello"。

第 14 行代码通过使用 padEnd(16, "Script")方法再次设定字符串目标长度为 16，然后在尾部插入字符串"Script"。

第 16 行代码通过使用 padEnd(17)方法再次从字符串尾部设定字符串目标长度为 17。

第 18 行代码通过使用 padEnd(18, "!")方法再次设定字符串目标长度为 18，然后在尾部插入标点符号"!"。

下面运行测试一下【代码 16-3】所定义的 HTML 页面，并使用调试器查看控制台输出的调试信息，页面效果如图 16.3 所示。

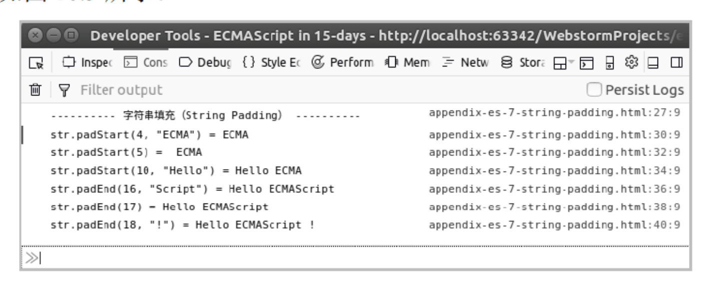


图 16.3　字符串填充（String Padding）

如图 16.3 所示，在通过使用 padStart()方法与 padEnd()方法进行一系列的操作之后，成功生成了字符串“Hello ECMAScript !”。

16.3.2 对象遍历

ECMAScript 8（2017）版本语法规范中新增了一组用于对象遍历的方法，分别为 Object.values()与 Object.entries()。这一组方法能够按照对象的键值对进行遍历，并将遍历结果返回为数组格式。

关于 Object.values()方法的基本语法格式如下：

```
Object.values(obj)
```

参数与返回值说明：

- obj：该参数是必需的，用于表示需要遍历的目标对象。
- 返回值：由“键值对”中的值所组成的数组。

关于 Object.entries()方法的基本语法格式如下：

```
Object.entries(obj)
```

参数与返回值说明：

- obj：该参数是必需的，用于表示需要遍历的目标对象。
- 返回值：由“键值对”所组成的二维数组。

下面看一个使用 Object.values()方法与 Object.entries()方法进行对象遍历的代码示例（详见源代码 appendix 目录中的 appendix-es-7-object-values-entries.html 文件）。

【代码 16-4】

```
01 <script type="text/javascript">
02 'use strict';
03       /*
04        * ECMAScript 8 (2017) 新特性 - 对象遍历
05        */
06       console.log("---------- 对象遍历 ----------");
07       const obj = {h : "Hello", e : "ECMAScript"};
08       console.log("-------- Object.values() --------");
09       console.log(Object.values(obj));
10       console.log(Object.values("ECMA"));
11       console.log("-------- Object.entries() --------");
12       console.log(Object.entries(obj));
13 </script>
```

关于【代码 16-4】的分析如下：

第 07 行代码定义了一个对象 obj，并进行初始化操作。

第 09 行代码通过使用 Object.values()方法对对象 obj 中的值进行了遍历操作，将会返回一个对象中“值”的数组。

第 10 行代码通过使用 Object.values("ECMA")方法直接对字符串"ECMA"进行遍历操作。注意，在直接对字符串进行遍历时，会先将字符串"ECMA"转换为对象格式{0:'E', 1:'C', 2:'M', 3:'A'}。

第 12 行代码通过使用 Object.entries()方法对对象 obj 中的值进行了遍历操作，将会返回一个对象中“键值对”的二维数组。

下面运行测试一下【代码 16-4】所定义的 HTML 页面，并使用调试器查看控制台输出的调试信息，页面效果如图 16.4 所示。

图 16.4　对象遍历

16.3.3　异步函数（Async Function）

ECMAScript 8（2017）版本语法规范中新增了异步函数（Async Function）的语法，具体是通过“async/await”来定义的。通过异步函数（Async Function）并结合使用 promise 对象，就可以很方便地实现异步函数功能。

其实，promise 对象是 ECMAScript 6（2015）版本语法规范中针对异步编程而新增的内容。虽然 promise 对象为异步编程带来一定的惊喜，但在状态控制与回调函数方面也有一定的缺陷。

技术总是在不断地进步，随着 ECMAScript 8（2017）版本语法规范中新增的异步函数（Async Function）的出现，将 promise 对象与异步函数（Async Function）结合起来使用，基本算是对 ECMAScript 脚本语言异步编程带来了本质上的突破。

在学习异步函数（Async Function）之前，先来了解一下 promise 对象的使用方法，有助于读者进一步理解异步函数（Async Function）的使用。

若要使用 promise 对象，须先创建一个 Promise 对象实例，基本语法格式如下：

```
/**
* 创建 Promise 对象
*/
const promise = new Promise(function(resolve, reject) {
  if (/* 异步操作成功 */){
    resolve(value);
  } else {
    reject(error);
  }
});
```

参数说明：

promise 构造函数接受一个函数作为参数，该函数的两个参数分别是 resolve 和 reject。注意，

这两个参数是两个函数，且是由 ECMAScript 引擎内置定义的。

resolve 函数的作用是将 promise 对象的状态从“未完成”变为“成功”，在异步操作成功时调用，并将异步操作的结果作为参数传递出去。

reject 函数的作用是将 promise 对象的状态从“未完成”变为“失败”，在异步操作失败时调用，并将异步操作报出的错误作为参数传递出去。

promise 对象实例生成以后，就可以用 then()方法分别指定 resolved 状态和 rejected 状态的回调函数，基本语法格式如下：

```
promise.then(function(value) {
  // success
}, function(error) {
  // failure
});
```

参数说明：

then()方法接受两个回调函数作为参数。第一个回调函数（必需的）在 promise 对象的状态变为 resolved 时调用；第二个回调函数（可选的）在 promise 对象的状态变为 rejected 时调用。

下面看一个使用 promise 对象执行异步编程的代码示例（详见源代码 appendix 目录中的 appendix-es-6-promise.html 文件）。

【代码 16-5】

```
01 <script type="text/javascript">
02  'use strict';
03       /**
04        * ECMAScript 6（2015）新特性 - promise 对象
05        */
06       console.log("---------- promise 对象 ----------");
07       /**
08   * function - runPromise
09        * @returns {Promise}
10        */
11       function runPromise() {
12           var p = new Promise(function (resolve, reject) {
13               setTimeout(function () {
14                   console.log('runPromise() done.');
15                   resolve('resolve data');
16               }, 1000);
17           });
18           return p;
19       }
20       /**
21   * call runPromise().then()
22        */
23       runPromise().then(function (value) {
24         console.log("call runPromise().then() : " + value);
25  });
26 </script>
```

关于【代码 16-5】的分析如下：

第 11～19 行代码定义一个函数 runPromise()。其中，第 12～17 行代码通过 promise 对象定义

一个对象实例 p，并通过 setTimeout()方法定义 1000ms 时长间隔的操作，执行第 15 行代码定义的 resolve()函数；第 18 行代码通过 return 关键字返回 promise 对象实例 p。

第 23～25 行代码通过 promise 对象调用 then()方法定义异步回调函数。

下面运行测试一下【代码 16-5】所定义的 HTML 页面，并使用调试器查看控制台输出的调试信息，页面效果如图 16.5 所示。

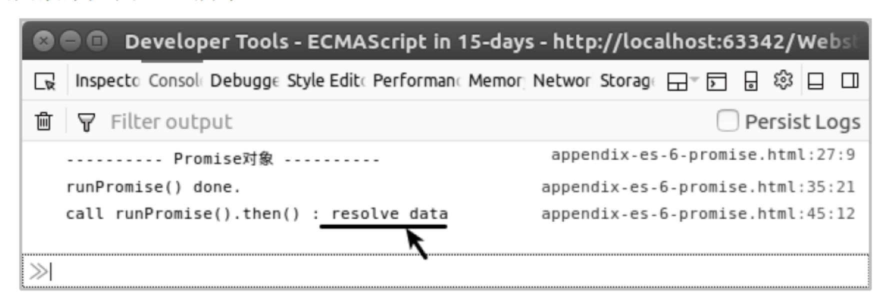


图 16.5　promise 对象异步函数

如图 16.5 中箭头所指，通过 promise 对象的异步编程操作，成功地异步获取了第 15 行代码定义的字符串 "resolve data"。

了解 promise 对象的使用方法后，就可以学习异步函数（Async Function）了。异步函数（Async Function）通过“async/await”定义，async 函数返回一个 promise 对象，并使用 then 方法定义回调函数。当函数执行时，一旦遇到 await 就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。

下面看一个异步函数（Async Function）执行异步编程的代码示例（详见源代码 appendix 目录中的 appendix-es-8-async-func.html 文件）。

【代码 16-6】

```
01 <script type="text/javascript">
02  'use strict';
03       /*
04        * ECMAScript 8 (2017) 新特性 - 异步函数 (Async Func)
05        */
06       console.log("---------- 异步函数 (Async Func) ----------");
07       /**
08        * function - runPromise
09        * @returns {Promise}
10        */
11       function runPromise() {
12           var p = new Promise(function (resolve, reject) {
13               setTimeout(function () {
14                   console.log('runPromise() done.');
15                   resolve('resolve data');
16               }, 1000);
17           });
18           return p;
19       }
20       /**
21  * async function - callPromise
22        */
```

```
23        async function callPromise() {
24           console.log("call runPromise()");
25           const externalFetchedText = await runPromise();
26           console.log("received : " + externalFetchedText);
27        }
28        /**
29   * procedure callPromise()
30         */
31        callPromise();
32        console.log("1st step...");
33        callPromise();
34        console.log("2nd step...");
35        callPromise();
36        console.log("3rd step...");
37 </script>
```

关于【代码 16-6】的分析如下：

这段代码是在【代码 16-5】基础上改写而成的，读者可以清楚地看到通过将异步函数（Async Function）与 promise 对象结合在一起实现异步函数的编写过程。

第 23～27 行代码通过 async 定义异步函数 callPromise()提供调用。其中，第 25 行代码通过 await 先返回，等到异步操作完成后，再接着执行函数体内后面的代码。

下面运行测试一下【代码 16-6】所定义的 HTML 页面，并使用调试器查看控制台输出的调试信息，页面效果如图 16.6 所示。

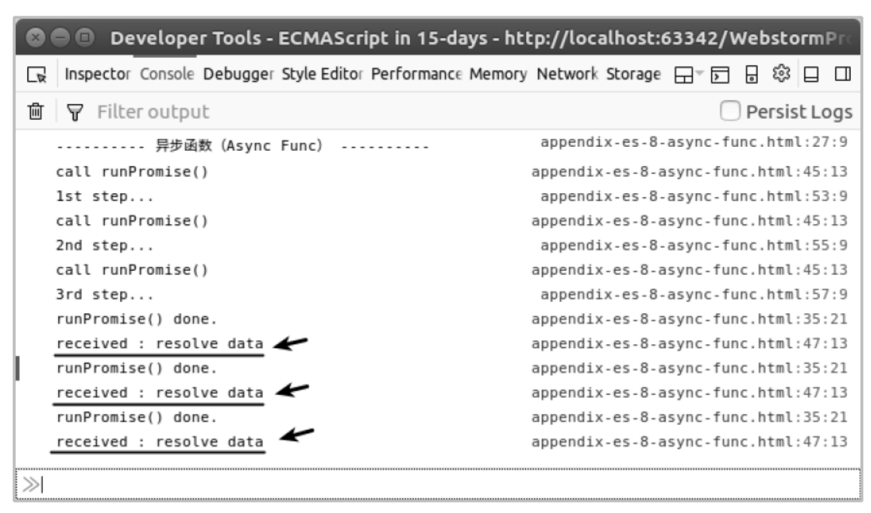


图 16.6　异步函数（Async Function）

如图 16.6 中箭头所指，通过将异步函数（Async Function）与 promise 对象结合在一起，成功地实现了异步编程操作。

16.4　本章小结

本章主要介绍了 ECMAScript 7 & 8 版本语法规范中新增的特性，包括字符串填充、对象遍历、异步函数等内容。这部分内容属于目前 ECMAScript 语法规范中的最新内容，且有一定的学习难度，相信通过文中给出的代码实例可以帮助读者加深理解这些新增的特性。